A REAL-TIME GARBAGE COLLECTION DESIGN FOR EMBEDDED SYSTEMS

Thesis

Submitted to

The School of Engineering

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree

Master of Science in Electrical Engineering

By

Joseph Thomas Fieler

UNIVERSITY OF DAYTON

Dayton, Ohio, USA

May 2004

A REAL-TIME GARBAGE COLLECTION DESIGN FOR EMBEDDED SYSTEMS

APPROVED BY:

Frank. A. Scarpino, Ph.D
Advisory Committee Chairman
Professor, Electrical and
Computer Engineering

John G. Webet, Ph.D
Committee Member
Professor, Electrical and
Computer Engineering

John S. Loomis, Ph.D
Committee Member
Associate Professor, Electrical and
Computer Engineering

Donald L. Moon, Ph.D
Associate Dean
Graduate Engineering Programs and Research
School of Engineering

Joseph E. Saliba, Ph.D
Dean, School of Engineering

Abstract

TITLE: A Real-Time Garbage Collection Design for Embedded Systems

NAME: Fieler, Joseph Thomas
University of Dayton

Advisor: Dr. Frank Scarpino

Digital hardware and software have gone through several revolutions during recent history. Hardware has progressed from vacuum tubes, to transistors, and finally to extremely powerful VLSI chips. Processors are designed with increasingly large instruction sets and memory widths and operate at ever rising speeds. Programming has progressed from machine code, to assembly language, to high level procedural languages, and finally to object oriented programming.

However some areas of computer architecture have changed very little since the modern computer's inception. Memory management is one of these areas. Accessing a computer's RAM is still accomplished through address and data lines. Managing the content of memory is entirely left to the programmer. The result is programs that either manage memory poorly by causing memory leaks, fragmentation, and systems failures or programs that utilize complex software memory managers that use a large amount of processor time.

This paper proposes that it is time for RAM to revolutionize the way it interacts with programs and how it treats the user's data. This next-generation RAM actively manages the dynamic information in its banks instead of being a passive device that blindly accepts addresses and data. The smart memory performs the same operations as the software memory managers that are used in programming languages such as Java. The main advantage that the smart memory has over traditional software memory managers is speed. Software managers need to execute several instructions and go through several layers of computer abstraction in order to manipulate the data. The smart

memory has immediate access to the data, does not execute on the main CPU, and runs at much higher speeds than software. The module also executes the memory management algorithm in real-time. This statement means that the smart memory module runs within a short, bounded, deterministic time. Real-time programmers traditionally restrain themselves from using dynamic memory allocation because its execution time is long and unbounded. The smart memory module provides real-time system designers with a quick and bounded way to manage dynamic memory.

The paper focuses on the theory of automatic dynamic memory management, commonly referred to as garbage collection (Chapter 1 and 2), the selection of an appropriate garbage collection algorithm (Chapter 2), and the implementation of the smart memory in hardware (Chapters 3 – 4). Chapter 5 lists the performance results of the smart memory module. Finally, Chapter 6 discusses applications, future avenues of research, and improvements to the smart memory module.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1

An Introduction to Automatic Dynamic Memory Management

Memory management is defined as how the programmer and system allocate and deallocate memory words. Allocating data involves finding a space in memory large enough to store the data, but not too large as to fragment memory. Deallocating memory is when old data is no longer need and the space it occupies is returned to the system. There are a variety of memory management options available for programmers to use. Three common types are static, stack, and heap memory [1].

Static memory allocates all storage for the program at compile time. No time is needed allocate data at run-time since all storage locations are predetermined. Real-time programmers often use static memory allocation because of its fast, deterministic allocation times. There are a couple of disadvantages to using static memory allocation. Only data that is allocated at compile time is stored. Arrays can not be dynamically lengthened at run-time to cope with an unexpected situation. Another disadvantage is that memory cannot be recycled once its data is no longer needed. Static memory allocation is quick at run-time, but inefficient in utilizing memory resources.

Stack memory allocates data using PUSH and POP commands. Little time is needed to allocate memory, but only data on top of the stack is immediately available. Accessing data on the bottom of the stack may take a long and unbounded amount of time. The big advantage of stack memory over static is that memory may be reused multiple times.

Heap memory allocation allows for truly random memory access. The program can address any word of memory at any time. Memory is reused and objects or arrays may be dynamically lengthened or shortened depending on the situation. The drawback to heap memory is the time-cost in managing the information. Time is needed to allocate

data and remove unused data. The allocation and deallocation of memory on the heap is called dynamic memory management since it is determined at run-time.

There are several ways to approach the problem of heap memory management. The easiest solution is not to remove the old data. This solution means that the heap becomes fragmented and full as the program allocates objects and does not remove dead ones. Obviously this method has many drawbacks. The program needs to be careful not to allocation more memory than is physically available. This restriction is impractical in most programs since it is hard to calculate the total memory usage, and some programs, such a real-time applications, cannot suffer an out-of-memory error on mission critical systems.

Programming languages such as the C and C++ leave the problem of heap management in the hands of the programmer. The software allocates memory in the heap through 'malloc' and 'new' commands and frees memory using 'delete' and 'free' instructions. Theoretically, this method presents an optimal solution to the problem of heap management. The programmer explicitly deletes a dead object from memory when it is no longer needed. The heap is always managed and there is never any dead data taking up memory space. In reality, explicit memory management does not function as perfectly as planned. Programmers may forget to delete obsolete objects. It is sometimes hard to know when to delete an object. This particular problem often arises in large programs with multiple programmers. One programmer might instantiate an object on the heap and expects it to be deleted by another programmer. However, the second programmer is unaware of the need to delete the object. It is difficult to tell when an object is truly out of scope and not needed in such large programs. Manual memory management defeats the modular notion of object-oriented programming because a programmer must know if an object allocates data on the heap that must be deleted outside of the original object.

The newer languages of Java and C# employ an automated system to manage the heap. This automatic dynamic memory management system is what is commonly referred to as garbage collection. The purpose of a garbage collector is to insure that a program has sufficient memory available to execute. Typically, a garbage collector scans the heap to determine whether an object is live or dead. Dead objects are reclaimed by

the system and the memory is made available for future use. The collector also compacts live data to insure sufficient free space for future large objects. Programmers use only the 'new' command when working with a garbage collected heap. The garbage collector takes the place of the 'free' instruction.

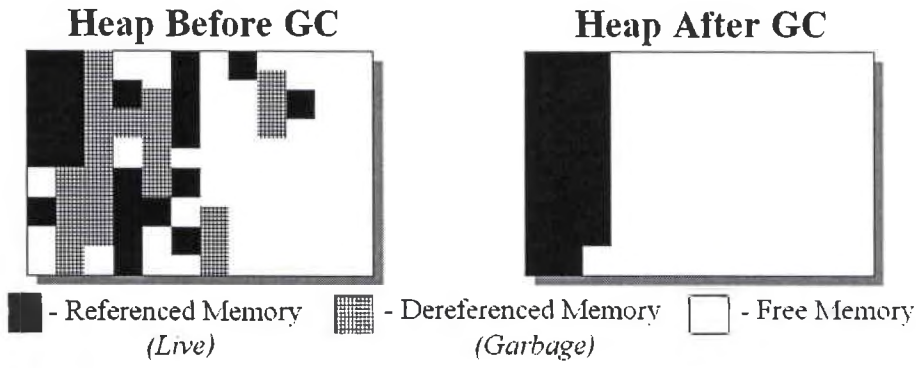

Figure 1.1: The Heap Before and After Garbage Collection

Figure 1.1 shows an example heap before and after garbage collection. The heap before collection contains fragmented live and dead data. The garbage collector returns the dead data to the free pool and defragments the live referenced data. The heap after collection is shown on the right of Figure 1.1.

1.1 Types of Garbage Collection

All garbage collectors perform the same task of managing the heap memory so that the program always has sufficient memory space. However, there are dozens of garbage collection algorithms to choose from. Each algorithm has its own set of advantages and disadvantages. There is no one garbage collection algorithm that is recognized as the all-around best. Generally, the system designer picks an algorithm that best suits their application. Most garbage collection algorithms fall within the classification of the three classical algorithms: Mark-Sweep, Reference Counting, and Copying.

1.1.1 Mark-Sweep

The first classical algorithm to look at is Mark-Sweep [2]. The algorithm is based upon a tracing routine. The tracing algorithm starts at the system roots, traces and marks all live objects, and then sweeps the unmarked (dead) objects to the free list.

Traditionally, the garbage collector is invoked when the system requires more memory than is available on the heap. The collector stops the mutator, performs the mark-sweep algorithm, and then restarts the mutator. All garbage has been removed from the heap by the collector.

Mark-Sweep, like most garbage collector algorithms, was originally developed to execute in software. Some software mark-sweep implementations make use of recursive subroutine calls to trace the live objects. However, recursive calling is slow, might cause system stack overflow, and is unsuitable for hardware implementation. Algorithms have been developed that eliminate the need for recursive mark-sweep algorithms. The non-recursive mark-sweep algorithms are based upon an auxiliary stack. [1] The auxiliary stack stores references to objects whose descendents need to be traced. The collector starts to sweep from the system roots. If an object that contains no references is found then it is marked black (live).[1] If the collector finds a live object that does contain references then it pushes the descendants onto an auxiliary stack. The collector pops objects of the stack one at a time, traces the descendent objects, pushes new descendent objects onto the auxiliary stack, and then blackens the original object. The next object on the stack is then popped and the process continues until the auxiliary stack is empty. Objects currently on the auxiliary stack are colored gray because they are live but still need to be scanned for descendants.

One problem with the auxiliary stack method is that it needs enough memory to store references to all the objects in the longest traceable path through the heap. This path can theoretically be as long as the total number of objects on the heap. If the heap has millions of objects then the auxiliary stack either needs a large amount of memory or risk overflow. The Boehm-Demers-Weiser algorithm [3] reduces the stack size, and makes accommodations for stack overflow. The stack, which is not as large as the number of objects on the heap, records when an overflow occurs. The collector stops pushing objects on the stack when the overflow happens. Objects on the stack are popped and blackened. The system then scans the heap for black objects that point to

[1] Colors are used to define an object's liveliness on the heap throughout the paper. The color black refers to objects that are found to be live and do not need to be revisited by the collector. The color gray means an object is live but needs to be scanned by the collector for descendants. White objects have yet to be reached by the collector, and all white objects at the end of a collection cycle are declared garbage.

white objects, and pushes those objects onto the now empty auxiliary stack. Scanning the heap is a time intensive process and should be done as few times as possible. The auxiliary stack size should be large enough to accommodate the vast majority of all traces. The Boehm-Demers-Weiser algorithm should be used only for the rare case of an extremely long trace.

1.1.2 Reference Counting

The second classical algorithm that is examined in this paper is reference counting. Reference counting is the only algorithm of the three classical garbage collection methods that is naturally incremental. An incremental algorithm is one that works along with the mutator to garbage collect memory in small steps. The other two classical garbage collector algorithms do not work in parallel with the mutator. They stop the mutator program, collect the heap memory, and then allow the mutator to resume. Reference counting performs incremental garbage collection whenever the mutator writes to a pointer.

The algorithm for reference counting garbage collection is very simple. Every object that is allocated on the heap contains a reference count field. The field keeps track of the number of references that are pointing to that object. Any object with a non-zero reference count is live. The algorithm is incremental in that every time the mutator writes to a pointer the reference counts of the effected objects are modified. If a reference count of an object drops to zero then the object is garbage and the memory is immediately reclaimed by the system. An example of the reference counting algorithm is shown below.

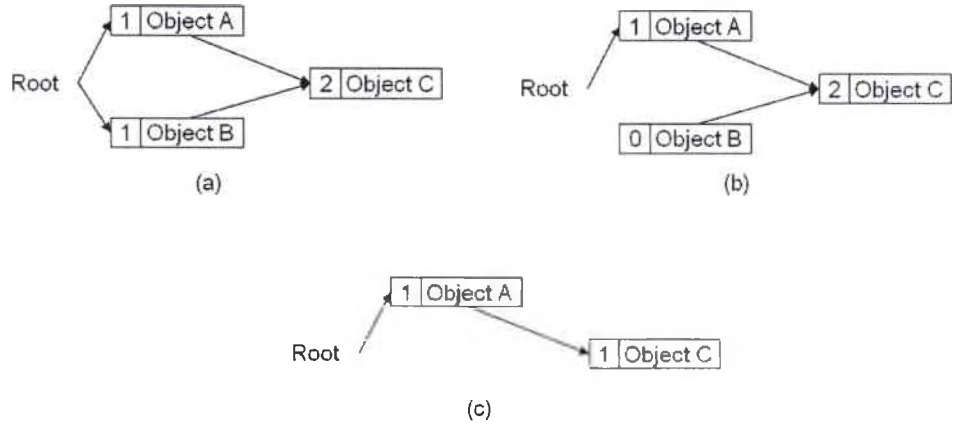

Figure 1.2: An Example of Reference Counting

Figure 1.2(a) shows object A and B are referenced by the system roots. Both of their reference counts are one. Both objects A and B also reference object C. In Figure 1.2(b) one of the system roots removes the reference to object B. The reference count of object B goes to zero and the object is immediately reclaimed by memory. Also, Figure 1.2(c) shows that object C has a reference count of one since it lost the reference from object B. The example above shows that the reference counting algorithm is intricately woven with the mutator program. What the garbage collector does next depends on the mutator's execution.

The main strength of the reference counting algorithm is that it is incremental. This aspect of the algorithm makes it an early contender for a real-time garbage collection algorithm. An incremental algorithm is preferred to a start-stop algorithm because it distributes the time costs of garbage collection over the entire execution of the mutator and does not stop the system for significant lengths of time. Another strength of the reference counting algorithm is that memory is freed to the system immediately when an object is found to be dead. Memory is reclaimed once the reference count of an object reaches zero, and that memory is available for instant use by the system. Finally, a garbage collector that is based upon a reference counting algorithm only needs to visit objects with modified references. The other classical algorithms are required to scan all live, and in some cases dead, objects. This scanning process is time and processor intensive.

Unfortunately, reference counting is not a perfect garbage collection algorithm. The method has several drawbacks. The most serious flaw in the reference counting

scheme is that it does not collect cyclic data [1]. Cyclic data is a set of objects that reference each other. Figure 1.3 shows an example of cyclic data on the heap.

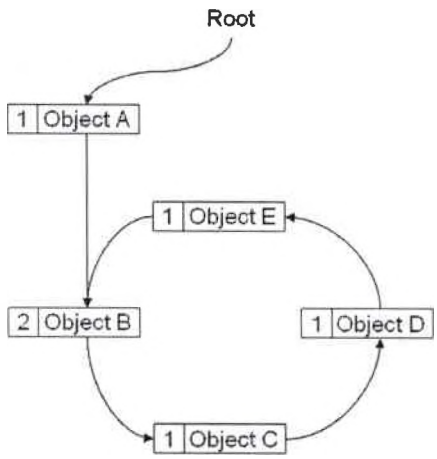

Figure 1.3: Cyclic Data on the Heap

Figure 1.3 shows that object B, C, D and E are part of the cyclic data structure. Take note that Object B has a reference count of two. The problem with the reference counting algorithm occurs when Object A deletes its reference to Object B. This step is shown below in Figure 1.4.

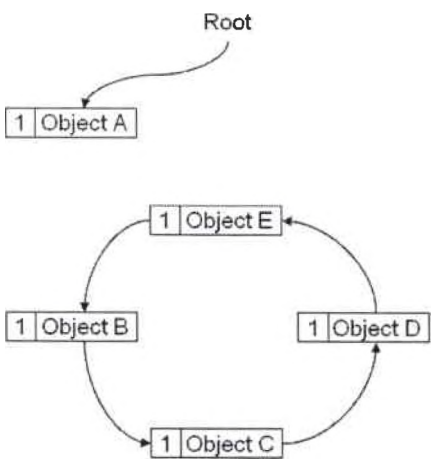

Figure 1.4: Dead Cyclic Data

Figure 1.4 shows that object B, C, D, and E are dead since they are no longer accessible from the root, but their reference counts are nonzero. The garbage collector still thinks the objects are live. The objects that make up the cyclic data structure are now a memory leak. They are of no use to the mutator, but their memory cannot be reused by the system.

There are ways to get a reference counting algorithm around cyclic data structures. The most common method is to use a hybrid reference counting/mark – sweep algorithm. The theory behind a hybrid collector is that the vast majority of objects are reclaimed by the reference counting portion of the collector, and any suspected cyclic data structures are collected by the mark-sweep collector. One hybrid collector was developed by Lin. [4] Lin uses four different colors to define objects in the heap. The color of each object is stored in the object's header. Black objects are known to be live. White objects are dead and have a reference count of zero. Purple objects are those that might be part of a cyclic data structure. The garbage collector colors objects purple when their reference count is decreased but does not go to zero. All purple objects are placed on a control set. The collector periodically performs a local mark-sweep on objects in the control set to determine if its nonzero reference count is due to an internal cyclic data structure. The last color, gray, tells the mark-sweep collector to revisit the object and to look for cyclic data structures. If an object is found to be part of a cyclic structure then it is reclaimed. Objects are lazily removed from the control set if the mutator somehow accesses them. Objects that are accessed by the mutator are obviously still live and in use.

Another drawback of the reference counting algorithm is the possibility of cascading deallocation. This problem occurs when there are a string of objects that reference one another in a sequential order [1]. Figure 1.5 shows a sequential chain of objects.

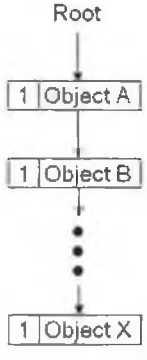

Figure 1.5: A Sequential Chain of Objects

When the first object in the chain is no longer referenced by the root then a cascade deallocation effect is started. The first object is reclaimed causing the next

object in the chain to have a reference count of zero. Then that object is reclaimed, and the process repeats itself. The mutator is stopped until the cascade effect reaches the last object. The worst case upper bound time of the entire process is the size of the whole heap. It is unacceptable to stop the mutator for this length of time.

An algorithm by Weizenbaum [5] is designed to lessen the impact of cascading deallocation. The algorithm does not immediately reclaim an object once its reference count goes to zero. Instead, the object is pushed onto a free list. The benefit of putting an object on the free list is that the reference to its descendents is not deleted. The descendents still have nonzero reference counts and are not reclaimed by the collector. The collector deletes objects from the free-list when time permits. Once the object is deleted then the object's descendents with only one reference are pushed onto the list. Weizenbaum does not actually prevent cascading deallocation. He merely distributes the execution time more evenly across the system.

1.1.3 Copying

The third and last classical algorithm is copying garbage collection. As the name implies, the heap is collected by copying only live objects from one area of the heap to another. The objects not copied are declared dead and their space is reused.

The most famous copying garbage collection algorithm is that of Cheney's [6]. He divides the heap into a FromSpace and ToSpace. The FromSpace is where all objects both live and dead, reside. The ToSpace is initially empty and is where the live objects are copied and newly allocated objects are stored. The figure below show a snapshot of the heap

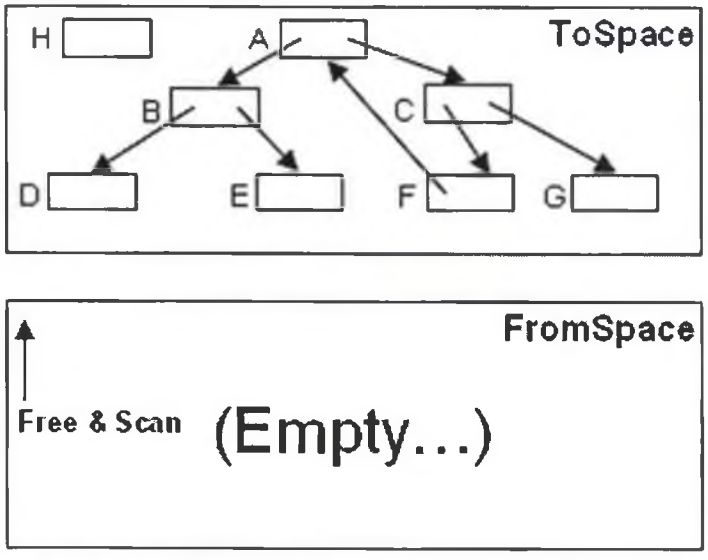

Figure 1.6: The ToSpace and FromSpace

The figure shows that the root points to object A and that object A points to several other live objects in the FromSpace. Objects H is not accessible from the root and is therefore a dead object.

The ToSpace is organized by a free and scan pointer. The two pointers are used to implicitly color all objects in the ToSpace. The free pointer marks the next location of available memory. The scan pointer marks the next object for the collector to scan for descendants. The free pointer always leads the scan pointer except at the beginning and end of a collection cycle. Objects that are located between the free and scan pointer are implied to be gray. Objects between the start of the ToSpace and the scan pointer are black. Black objects are live data and gray objects are live data that need to be scanned for descendants. An example of Cheney's copying algorithm starts with the heap organized as shown in Figure 1.6.

The collector scans the heap starting from the root. Any object that is reachable from the root is live. An object is copied to ToSpace when it is traced by the collector. A forwarding pointer is written in the object's old FromSpace location so that system knows the object's new address. The free pointer is then advanced to the next location of available memory. Because the newly copied object is between the free and scan pointer it is implied to be colored gray. The gray object that is referenced by the scan pointer is then searched for descendants. The descendants are copied into the ToSpace, the scan

and free pointers are advanced, and the original gray object is now colored black. The copy collection process repeats itself until the free and scan pointers point to the same memory address. All objects that are not copied are garbage. New objects are allocated at the memory address of the free pointer. The garbage collection cycle is restarted when the ToSpace runs out of memory. The ToSpace and FromSpace swap titles and the collection process is repeated.

Figure 1.7 shows an example of copying garbage collection in action.

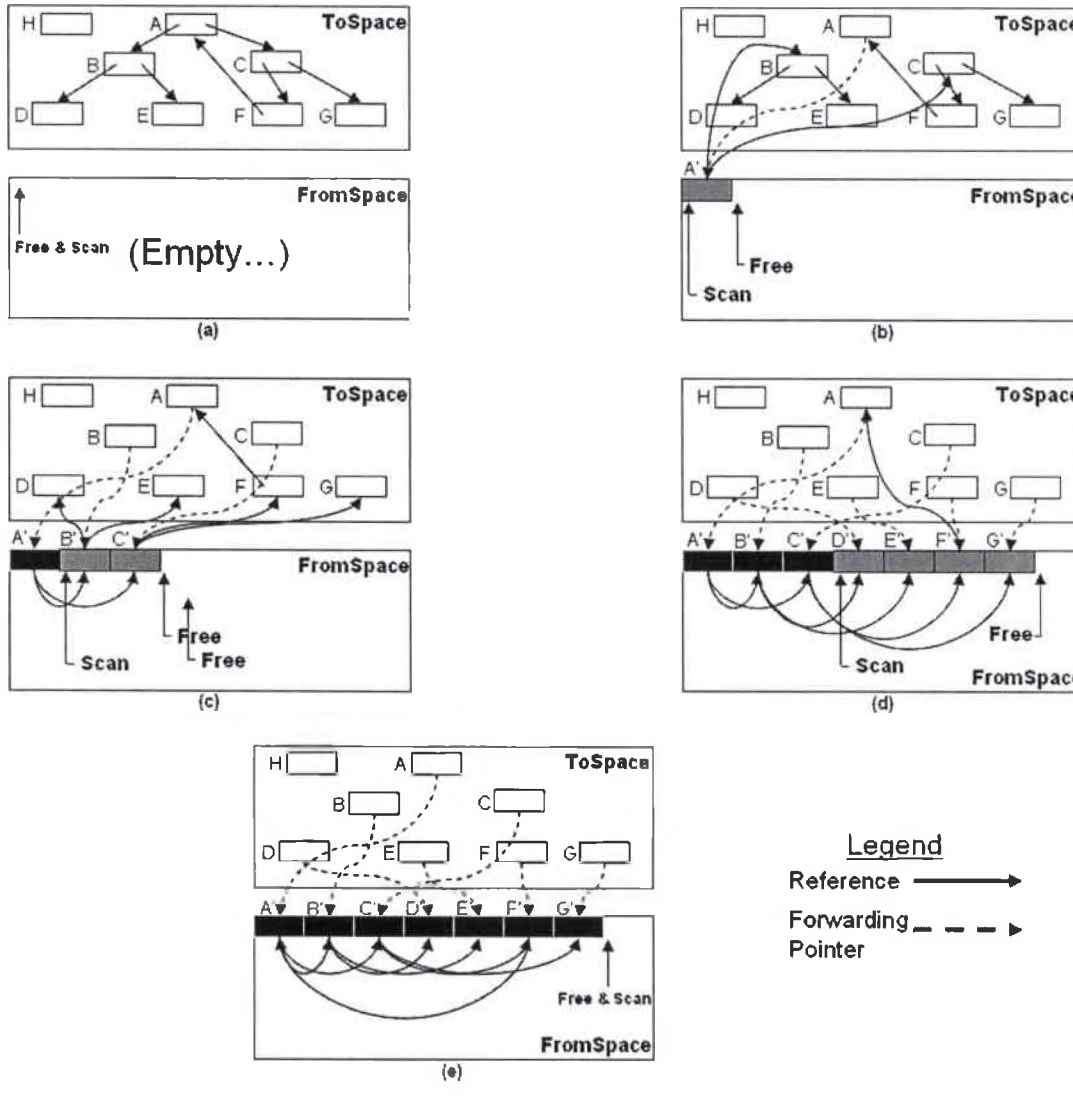

Figure 1.7: Copying Garbage Collection in Action

The first snap-shot of the algorithm 1.7(a) is identical to that of Figure 1.6. There are eight objects in the heap and one root pointer. The algorithm starts by copying object A to the ToSpace. The free pointer is advanced and object A is recognized as gray in

step 1.7(b). Note that a forwarding pointer is written in Object A's old FromSpace location. The reference points to the object's new location in ToSpace. Object A is then scanned for descendants in 1.7(c). Objects B and C are descendants of Object A and are copied to the ToSpace. The scan pointer is now incremented to the next gray object and Object A is colored black. Objects B and C are colored gray because they reside between the Scan and Free pointers. Object B and C are scanned for pointer and their descendants are copied in Figure 1.7(d). Figure 1.7(d) shows that Object F in still contains a reference to Object A's old location in ToSpace. This reference error is corrected when Object F is scanned for descendants. The collector realizes that a forwarding pointer exists in Object A's old FromSpace location, and changes the reference to Object A's new location in ToSpace. This is shown in Figure 1.7(e). Object H is unreachable from the root, not copied, and is therefore dead. The collection cycle is over when the Scan and Free pointers are equal as they are in 1.7(e). Any newly created objects are allocated at the free pointer (the pointer is then advanced).

Copying garbage collection has many strong points. It is only required to visit live data. Mark-Sweep collectors need to visit dead objects during the sweep phase. The only overhead that is required by the collector is the scan and free pointers. Both of these pointers are only 32-bits in size. In contrast, reference counting algorithms require a memory word on every object for the reference counts, and mark-sweep algorithms need coloring bits for every object. The copying algorithm is extremely quick at object allocation. New objects are just allocated at the free pointer. Putting new objects on the heap in this manner is as costly to pushing objects on a stack. No processor time is needed to find a spot of free memory large enough to fit the new object. The copying method also naturally defragments the heap memory during collection. This compaction increases the efficiency of memory utilization by not populating the heap with small, unusable blocks of free memory.

There are a few drawbacks to using the copying garbage collection algorithm. The main problem is that the method requires twice the memory space in order to have both a ToSpace and a FromSpace. This issue is becoming less of a problem as the price of memory keeps dropping significantly. The next problem is that the algorithm is not efficient with large live objects. This issue is because every live object must be copied to

the ToSpace. Memory copying is a time expensive procedure. If an object is several thousand or million bytes in size then the system must wait for the whole object to be copied. Copying garbage collection works best with small, short lived objects.

1.2 Summary

The three basic garbage collection algorithms show that no one method solves all the problems that automatic memory management presents. Below is a table describing the main advantages and disadvantages of the algorithms.

Table 1.1: A Comparison of Garbage Collection Algorithms

	Mark-Sweep	Reference Counting	Copying
Positives	• Collects cyclic data structures • No overhead on pointer manipulation	• Incremental • Instantly Frees Dead Memory	• Quick object allocation • Only traces live data • Automatically compacts heap
Negatives	• Must visit both live and dead objects • Complex object allocation	• Does not reclaim cyclic data • High overhead on pointer manipulation • Cascading deallocation • Complex object allocation	• Poor performance for systems with large objects • Requires twice as much memory

The conclusion that one can draw from Table 1.1 is that the selection of a garbage collection algorithm depends on the system, programming language, and the type of programs being run on the machine. A system that does not use cyclic data and contains objects directly referenced by the root would work well with a reference counting collector. Systems with small, short lived objects work best with copying collectors. Mark-sweep collector are good for systems that have complex data structures, require the objects to not move in memory, and cannot afford the extra cost of a separate ToSpace and FromSpace. The next chapter analyzes real-time algorithms that are derivatives of the three classical algorithms.

Chapter 2

Real-Time Garbage Collection

The previous chapter discussed the three main algorithms for garbage collection. The past chapter shows that each algorithm has both strong and weak points, and that no one algorithm is universally recognized as the best garbage collection method. The same statement is true with real-time garbage collection algorithms. This chapter examines the definition of a real-time garbage collection algorithm, three real-time garbage collection techniques, and the reasoning behind the garbage collection algorithm that is chosen for implementation.

The first step in selecting a real-time algorithm is defining the meaning of real-time. There are two versions of real-time popular in the garbage collection community: soft real-time and hard real-time [7, 8]. Soft-real time tries to minimize the impact of the collector on the mutator by demanding that garbage collection be completed in a reasonable amount of time. However, the collection process is not bounded. In general, a soft-real time system expects results within the time allotted, but prefers late results over no results at all. Examples of soft-real time systems are streaming internet video and multitasking operating systems. Hard real-time systems demand that all of the collector's actions take place within a bounded time period. A hard real-time algorithm is fast, bounded, and predictable. This type of real-time system considers late results just as wrong as incorrect results. Examples of hard-real time systems are aviation flight computers and factory controls. The garbage collector that is designed for this thesis follows the hard-real time algorithm.

The mark-sweep and copying garbage collectors discussed in the previous chapter are classified as start-stop algorithms. These methods stop the mutator while the collector cleans the entire heap. The mutator is allowed to continue only after the entire

heap has been collected. The amount of time that the mutator is paused for is often too long for hard real-time systems. Therefore, most real-time garbage collection algorithms allow for interleaving both the mutator and the garbage collector. The management of dynamic memory is distributed out over small, incremental steps as opposed to the single, large pause time of the start-stop algorithms.

2.1 Read and Write Barriers

The synchronization of the mutator and collector reduces the pause time that is imposed by the collector, but also creates the problem of two systems accessing and modifying the same data. An example of this problem is shown below.

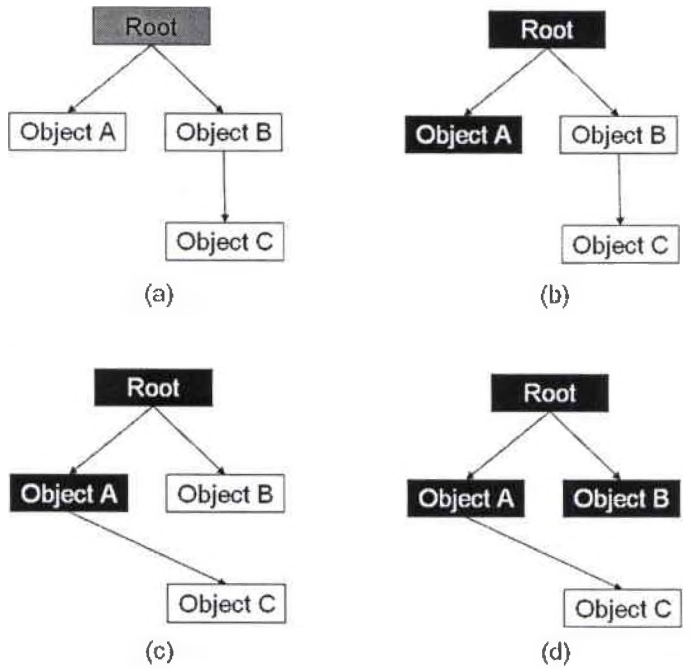

Figure 2.1: Mutator/Collector Interaction

Figure 1 shows the heap at various stages of mutator and collector execution. Step 2.1(a) shows that the root references objects A and B and that object B references object C. The memory manager performs an incremental collection between steps 2.1(a) and 2.1(b). Step 2.1(b) shows that object A has been marked as live and that it has no descendants. Therefore, the object is colored black.[2] The mutator runs between steps

[2] The color black denotes objects that are live and no longer need to be visited by the collector. The color gray signifies objects that the collector knows are live needs to revisit. Objects of the color white have not been visited by the collector, and are declared garbage at the end of each collection cycle.

2.1(b) and 2.1(c) and changes the reference layout of the heap. Object A now references object C, and object B's reference to object C has been removed. Unfortunately, the collector thinks that it no longer needs to visit object A to check for descendants. The collector runs one last time between steps 2.1(c) and 2.1(d). Step 2.1(d) shows that object B is marked as live and colored black because it no longer has descendants. The collector believes that it is finished with the current collection cycle, but object C is marked as dead even though it is really live [1].

The key to preventing the mutator from catastrophically modifying the heap's data during collection is to prevent black objects from having references to white objects. Figure 2.1 above shows that the problem begins when the mutator stores a reference to the white object C in the black object A. There are two methods to prevent the problem of black objects seeing white objects. The first is called a write barrier.

Write barriers are used predominately in real-time mark-sweep algorithms. The real-time mark-sweep collectors scan X amount of objects during each collection pause instead of the entire heap. Sweeping of the heap is relatively fast and usually occurs during only one pause. The write barrier is what prevents the mutator from changing the structure of the heap's data without the collector being aware of the change. There are several different types of write barrier algorithms. All prevent black to white references, but differ in form of execution and efficiency. Like garbage collection algorithms, no one write barrier method is widely considered the best. The best algorithm depends on factors like the program language. Below is a few of the many write barrier algorithms.

2.1.1 Yuasa's Write Barrier

The first write barrier to be examined is Yuasa's snapshot algorithm [1, 9]. Yuasa enables his write-barrier by catching writes to existing pointer fields. A white object is immediately colored gray and pushed onto the auxiliary stack if a reference to it is overwritten. Going back to Figure 2.1, Yuasa's write barrier immediately colors Object C gray when the reference in Object B is deleted. Furthermore, Object C is then pushed onto the auxiliary stack so that it is scanned for descendants. The auxiliary stack is discussed in Chapter 1. Object C and any of its descendants are saved from being inadvertently deleted.

Yuasa's algorithm is extremely conservative in that even dead objects survive for one extra garbage collection cycle. The write barrier still colors an object gray when its last reference is deleted. The dead object is finally removed in the next garbage collection cycle when there are no references left to trigger the write barrier. Yuasa's write barrier is referred to as a snap-shot algorithm because any change to the heap structure is noted and any object that is live at the beginning of a collection cycle still exists at the end of collection.

2.1.2 Dijkstra's Write Barrier

Another popular write barrier is Dijkstra's incremental algorithm [10]. Dijkstra's write barrier works by coloring a white object gray whenever a reference is made to that white object. Dijkstra's algorithm is different then Yuasa's in that it does not preserve objects that lose pointers. Dijkstra guarantees that objects are retained if they gain pointers. Dijkstra's write barrier has the advantage of collecting garbage in the same collection cycle that an object dies. The one problem with Dijkstra's write barrier is that it can produce floating garbage. This garbage occurs when a reference is written to an object, the object is colored gray, the object later loses all of its references, and then the object becomes garbage but is not collected in that cycle. The dead object is deleted in during the next collection cycle.

2.1.3 Read Barriers

The other method to preventing black objects from referencing white objects is a read barrier. This type of barrier is used in copying collectors. The read barrier automatically collects a white object when the mutator reads a reference to it. The theory is that a white object cannot be referenced by a black object without the reference first being read by the mutator. The collector detects the read and collects the white object.

The reason why read barriers are necessary for copy collectors is because two systems, the mutator and collector, are both modifying the heap structure. For example, say the copy collector moves an object from the FromSpace to the ToSpace. Later, the mutator reads a reference to the old location of the object, and then uses data that is stored in the object's old location. The problem is that the data in the old location is

obsolete and may cause the mutator to malfunction. Write barriers do not catch this error since they only check memory writes from the mutator. Read barriers do catch this error since they check memory reads instead.

Both write and read barriers impose processing costs on the system. The barrier that is the most efficient in a system is the barrier that is used the least. If a system reads data less often than it writes data then the read barrier is more efficient. A write barrier is more efficient if there are more read commands than write commands. It is generally believed that read commands outnumber write commands in most systems. That is why most mark-sweep collectors use write barriers instead of read barriers. Write barriers are used in non-moving algorithms since there are never references to old, obsolete objects. Copy collectors do not have a choice between barriers for the reasons described in the paragraph above.

2.2 Real-Time Garbage Collectors

2.2.1 Baker's Copying Collector

One of the most popular real-time garbage collecting algorithms is Baker's Copy Collector [11]. Baker's algorithm is similar to Cheney's copying method that is described in Chapter 1. There is still a ToSpace and a FromSpace. Live objects in the FromSpace are copied and scanned in the ToSpace. Only live objects are traced and the heap is automatically compacted. The difference between both algorithms is in object allocation. Cheney's algorithm allocates objects at the Free pointer. Baker's real-time algorithm allocates new objects at the end of the heap using a New pointer.

Cheney's method of allocating new objects at the Free pointer works well for start-stop collection. The problem with allocating at the Free pointer starts when collection is done incrementally. Objects that are allocated at the Free pointer are automatically colored gray. This color means that the collector still needs to scan the objects for descendants that are in the FromSpace. However, the read barrier prevents new objects from having descendants in FromSpace. The end result is that precious time is spent scanning the new objects for something that is never there.

Baker's real-time algorithm avoids this problem by allocating objects at the end of the heap. The new objects are colored black on creation and no extra time is wasted in scanning the objects for something that does not exist. The read barrier prevents the new objects from being initialized with references to FromSpace or having those reference written to them during the collection cycle. Figure 2.2 shows the layout of the ToSpace for the Baker system.

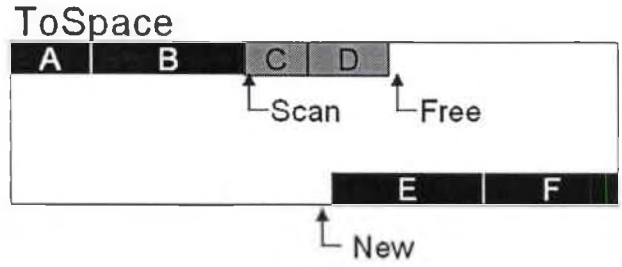

Figure 2.2: Baker's Real-Time Copying Collection

Figure 2.2 shows that the top of the heap in Baker's algorithm is identical to Cheney's algorithm. Baker's method is different in that new objects are allocated from the New pointer. It is also noted that new objects are allocated black. This definition means that the collector does not have to scan new objects. The definition also means that dead objects cannot be collected during the same collection cycle that they were created since they are automatically colored black. This is similar to the problem of floating garbage that is produced by Dijkstra's write barrier.

Baker's real-time copying garbage collection provides an elegant algorithm to collect the heap. Only live objects are scanned, and the heap is automatically compacted. Allocation is fast and similar to pushing objects on a stack. The algorithm requires only three pointers to distinguish between live, dead, and new objects. In contrast, mark-sweep algorithms require an auxiliary stack to scan the heap, the auxiliary stack might overflow, and both live and dead objects are visited during collection. The main problem with Baker's real-time copying collection is that the delay time is bounded to the size of the objects. Objects several thousand bytes in size are going to cause a large delay in the mutator's real-time execution. A real-time system that employs Baker's copy collector either needs to limit the size of objects or utilize hardware acceleration to decrease copying time.

2.2.2 Treadmill Garbage Collector

Henry Baker has also developed a non-copying garbage collection algorithm [12] that maintains several of the advantages of the copying algorithm. It allows for quick pointer allocation, but does not include the copying overhead of Baker's previous algorithm. The new algorithm is referred to as Baker's Treadmill Collection. The Treadmill initializes the memory by forming a doubly-linked circular list of memory blocks. One word from the memory block is used for the forward pointer and another word is reserved for the reverse pointer. The figure below shows the layout of the memory blocks.

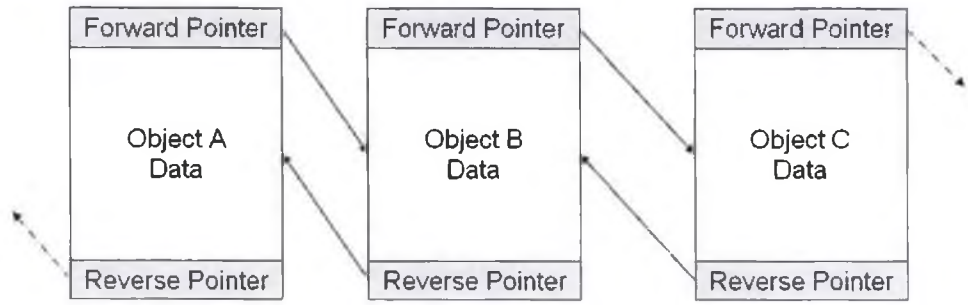

Figure 2.3: Doubly-Linked Memory

Each object can be no larger than one block of memory. The collector classifies objects by four colors. Black objects are live and scanned objects. Gray objects are live but still need to be scanned for descendants. Off-White cells are objects in the FromSpace. Finally, white objects are part of the free list. The colors are separated by four pointers. The pointers are Bottom, Top, Free, and Scan. The Bottom pointer separates white and off-white colors. The Top pointer separates the off-white and gray objects. The Scan pointer separates the gray and black colors. The Free pointer separates the black and white objects. Figure 2.4 shows the layout of the objects and pointers.

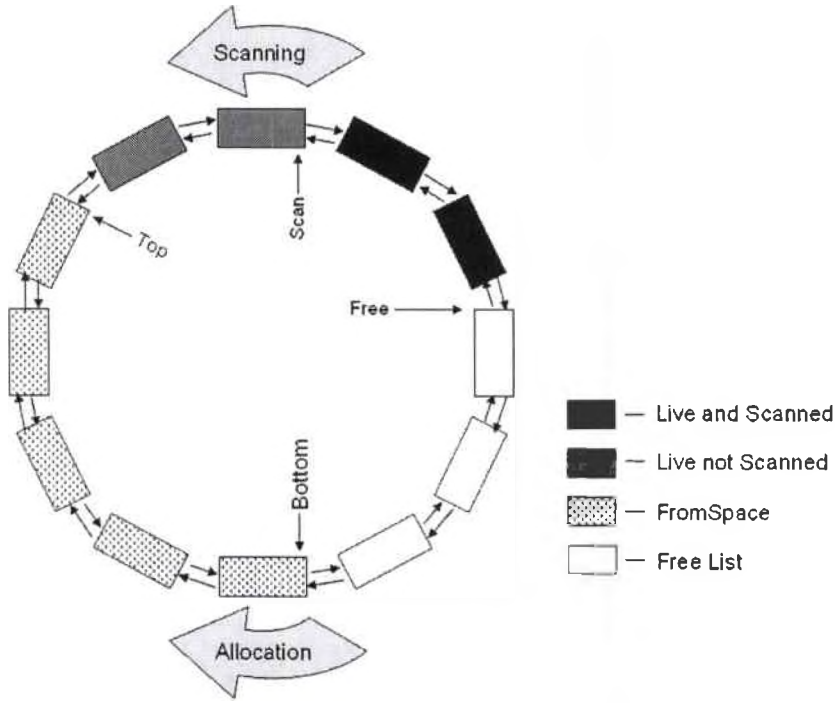

Figure 2.4: The Circularly Linked List and Pointers [1]

The figure above shows the circular linked list, the pointers, and the objects to which the pointers are assigned. The algorithm starts by allocating all objects white (free). New objects are allocated at the Free pointer. The Free pointer is then moved clockwise to the next free memory block in the list. The new object is classified as the color black because it now resides between the Scan and Free pointers. Allocation in the treadmill has low overhead because the new object is allocated at the Free pointer and then the Free pointer is moved to the next block in the list. This example is in contrast to potentially long and unbounded time of mark-sweep allocation. The only drawback to treadmill allocation is that dead objects cannot be collected during the same collection cycle that they are created because new objects are automatically colored black at creation.

The Scan pointer denotes the next object that is live and needs to be scanned for descendants. The designated object is scanned during a collection cycle. The Scan pointer is then moved counter clockwise to the next gray object. The freshly scanned object is now black because it resides between the Scan and Free pointer. Descendants of the object are moved from the ToSpace portion of the linked list gray section of the list between the Top and Scan pointers. This movement is the only time in the Treadmill

algorithm where the forward and reverse pointers of a memory block are changed. The same operation in Cheney's copy collector requires the entire object's data to be physically moved from the FromSpace to ToSpace. The treadmill accomplished the same task by just changing two pointers.

The algorithm does require one color bit to define whether the object is in the off-white portion of the list. This bit is for the collector to determine whether the descendants of an object are in the FromSpace or the ToSpace. Descendants in the FromSpace are moved and colored gray. Descendants in the ToSpace are left alone.

A garbage collection cycle lasts until the Scan and Top pointers are equal (i.e. no gray objects are left). At this point any objects left in the FromSpace are garbage and moved to the Free list. The objects that are colored black are moved between the Top and Bottom pointer to form the new FromSpace. There are now no objects in the ToSpace. The new collection cycle now starts and repeats the same steps that are discussed in the paragraphs above. Note that the contents of objects are never actually moved. Only the pointers that define the linked list are modified.

The collector allows for interweaving the execution of the mutator and garbage collection. The algorithm prevents the mutator from damaging the heap during collection by utilizing a read-barrier and allocation is handled simply and quickly by the Free pointer.

There is one large problem with the treadmill collector. The problem is that the memory blocks that make up the doubly linked list also limit the size of objects. An object cannot be larger than the size of one memory block. One could make the memory blocks oversized to accommodate large objects, but this solution wastes memory space because only one object is assigned to each memory block and very small objects would also use the oversized memory block. The other solution is to restrict the maximum size of objects, but this way out confines the programmer and may make the module unusable for applications such as image processing.

2.3 Selection of a Real-Time Garbage Algorithm

The previous section described three methods of real-time garbage collection. There was the mark-sweep algorithm with a write barrier, the Baker collector, and the

Treadmill method. The benefits and drawbacks of each system were described in detail. Table 1 shows a tabulate form of the pros and cons for each type of collection

Table 2.1: Three Types of Real-Time Garbage Collection

Garbage Collection Type	Positives	Negatives
Mark-Sweep (write barrier)	• Write barrier is less time consuming than read-barrier • Variable sized objects • Objects are stationary	• No pointer allocation • Requires algorithm to find needed space for objects • Memory fragmentation
Baker's Copying	• Pointer allocation • Automatically compacts data • Variable sized objects	• Large pauses when copying large objects • Requires twice as much memory • Requires handle pool
Treadmill	1. Objects are stationary 2. Pointer allocation	3. Limit on object size 4. Inefficient memory utilization

The mark-sweep collectors are attractive for a number of reasons. They use a write barrier instead of a read barrier. The write barrier is more efficient because it is invoked less often than a read barrier. There are no restrictions on object size like there are in the Treadmill algorithm. The big advantage that mark-sweep has is that it does not move data. The movement of data is time consuming and forces the use of a read barrier. However, the mark-sweep algorithms do have a number of disadvantages. Memory fragments as objects die. A defragmenter is possible to implement but doing so eliminates the collector's main advantage of fixed data. Object allocation is potentially a long and unbounded process. Algorithms are needed to search the fragmented heap for a suitable location for new objects. It is this unpredictable allocation time that disqualifies the mark-sweep collector from contention for implementation into hardware.

The treadmill algorithm tries to combine both the non-moving data advantage of mark-sweep along with the pointer allocation benefit of Baker's copy collector. The treadmill uses a doubly linked list to accomplish this goal. New objects are allocated at a location indicated by the New pointer, and FromSpace objects are transferred to the

ToSpace by changing just two pointer values. Unfortunately, the linked list is also what eliminates the collector from real-time hardware implementation. The linked list requires that all objects fit within a predefined memory block of a certain size. Limiting objects to a small size inhibits programmers, but allowing large sized means that much of memory is left empty and wasted.

Baker's Copying collector has the advantage of pointer allocation and automatically compacting memory. The one disadvantage is the amount of time that is required to copy data. However, the amount of time that is required to copy an object is short, bounded, and predictable with hardware acceleration. New object allocation with the scan-sweep method is always going to be long and unpredictable with or without hardware acceleration. The Treadmill algorithm is always going to restrict the size of an object with or without hardware acceleration. Baker's copy collector is chosen for hardware implementation because of short allocation time, automatically compacting memory, and its one drawback, copying data, is minimized with hardware implementation.

Chapter 3

Design of a Hardware Garbage Collection Module

The previous two chapters devote a great amount of time to garbage collection algorithms and the impact that a real-time, incremental system has on memory management. Chapter 1 details the three classical automatic memory management algorithms. Chapter 2 ends with the selection of Baker's Incremental Copy Collector as the collector of choice for this system. The purpose of Chapter 3 is to describe the design of the collector, how it interfaces with the system, and it hardware implementation.

Garbage collection provides the obvious benefit of a stable, reliable system. The collector prevents against memory fragmentation and automatically reclaims any memory space that is used by dead objects. The main case against automatic dynamic memory management is that it slows program execution. Programs that use early versions of garbage collectors usually spend forty percent of their time collecting the heap. Modern software collectors use around ten to twenty percent of the program executions time [1]. This percentage is still too high for people using real-time systems. Furthermore, collection is often long, unbounded, and unpredictable. The purpose of this paper is to make garbage collection quick, bounded, and predictable through hardware acceleration. This chapter examines hardware implementation of the garbage collector for the purpose of improving collector execution time.

3.1 Background

The hardware garbage collector is described in the beginning of the paper as a smart memory module. This statement correctly implies that the final hardware design is to be incorporated into a memory module. However, there are other possible locations for the collector. Specialized hardware can be inserted into either a computer's CPU, a

chip on the motherboard, or in an expansion card. The problem with these locations is that they execute slower than a collector inside the memory module executes. The CPU needs to fetch and execute instructions from memory. The collector also needs to share processor time with other applications. The motherboard and expansion board alternatives require the use of buses to reach the memory. These buses may not always be available since other devices in the system use them too. The need for speed is paramount in a real-time application. Therefore, it is decided that the garbage collector is to reside in the memory module. Hardware in a memory module has immediate access to the data. This choice also eliminates the need to share processor time and the uncertainty of bus availability.

3.1.1 Kelvin Nilsen's Garbage-Collecting Memory Module (GCMM)

Research into a garbage collection memory module has already been done by Kelvin Nilsen of Iowa State University. [13, 7] The goal of Nilsen's research is to design a cost-efficient, real-time garbage collector in hardware. He envisions a smart memory module that interacts with standard memory buses. He calls his design the Garbage-Collecting Memory Module (GCMM). The only difference between his memory and regular RAM is that the smart memory module also performs garbage collection. A figure describing his concept is shown below.

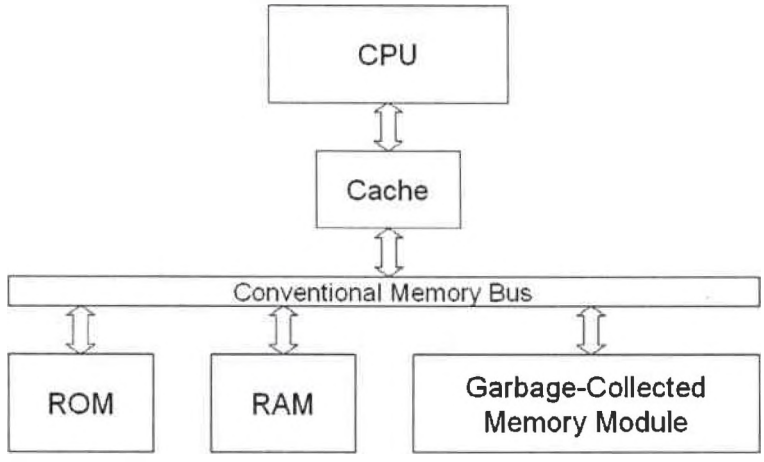

Figure 3.1: Nilsen's GCMM Concept

Nilsen chose to locate his collector in memory for speed and flexibility. He envisions his GCMM being used in machines with different CPUs. Nilsen states that the

GCMM is CPU independent and can operate on any machine with only small changes. The GCMM contains memory, internal buses, and the hardware needed to perform garbage collection. Figure 3.2 shows the inner workings of the GCMM.

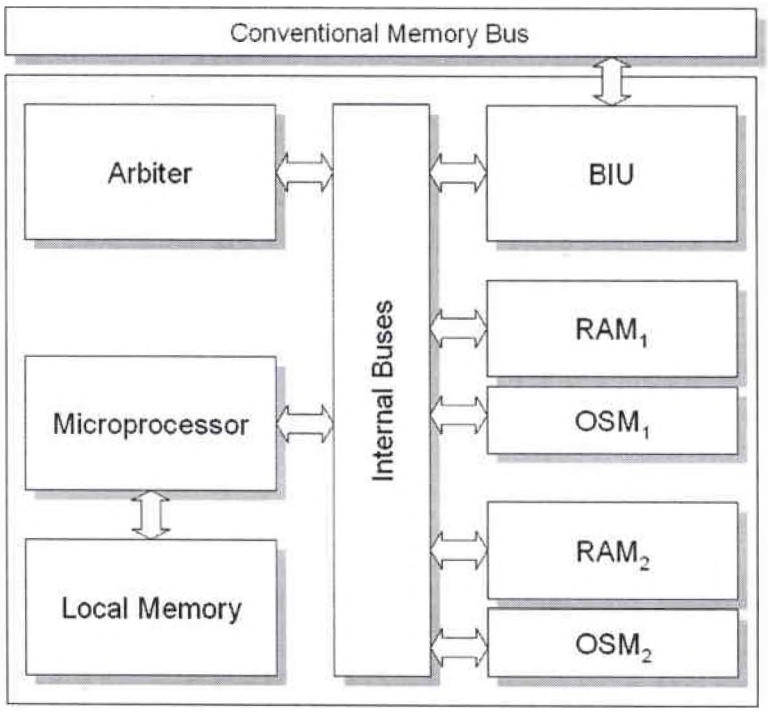

Figure 3.2: The Internal Design of the GCMM

The bus interface unit (BIU) lets the system communicate with the GCMM. The data is stored in the two RAM modules. The RAM modules are on separate buses to allow for parallel access. Also, the width of the memory word is 33 bits. The extra bit allows the collector to tag references. The object space managers contain the starting location of each object in their respective RAM chips. The microprocessor performs the actual garbage collection. Its program is stored in the local memory. The arbiter controls the access to the RAM modules. Requests from the system through the BIU are given the highest priority. Requests from the garbage collecting microprocessor are given low priority. The result is that garbage collection is treated like a background task in Nilsen's design [13, 7].

The algorithm that is executed in the GCMM is a version of Baker's copy collector. The two RAM chips are used as a ToSpace and FromSpace. Nilsen uses lazy

copying for the larger objects. This means that space is reserved for the large objects in ToSpace, and the object is incrementally copied whenever there is free time in the system. This algorithm prevents a large object that is being copied from holding the system for a long time. Nilsen states that his design can allocate a new object within 2μs, fetch a word in 2μs, and read a word in 500ns.

3.2 High-Level Design

Many of Nilsen's ideas are incorporated into the smart memory module (SMM). The first idea is embedding the garbage collector inside the memory module. The SMM also utilizes Baker's copy collector. The version that is implemented in this design does not use lazy copying. Lazy copying is effective at spreading out the cost of copying large objects over time, but does not eliminate the problem. The smart memory module is currently not meant for large objects. Large objects such as images should be stored in conventional RAM to improve system performance. Chapter 8 theorizes on ways for large objects to be copied in a small amount of time.

Below is an image of the smart memory module.

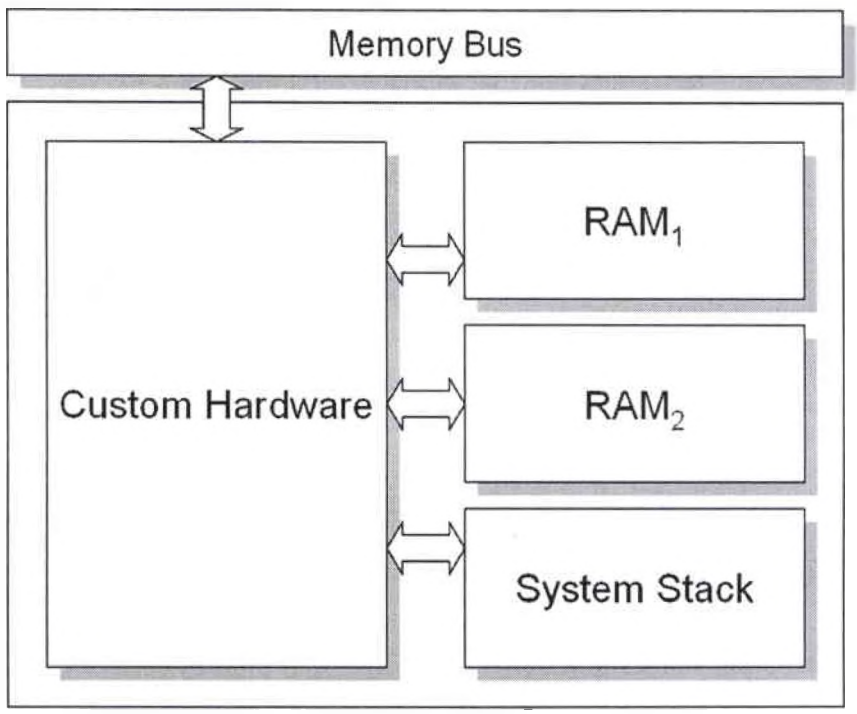

Figure 3.3: The Smart Memory Module (SMM)

Figure 3.3 shows that while there are similarities between the SMM and GCMM there are also many differences. The first difference is how the system interacts with the smart memory module. Nilsen's design treats garbage collection as a background task to run whenever there is no instruction from the mutator. Nilsen's arbiter assigns garbage collections task a low priority and gives high priority to mutator requests. The smart memory module is built specifically for a hard real-time system and runs when instructed for a predetermined amount of time. The real-time scheduler sets aside a certain amount of time for the collector during a cycle, and the smart memory collects the heap during that period. It is important to note that the smart memory does not exceed its time limit. Programming languages such as Java include a garbage collection instruction. This instruction informs the memory of when to collect.

The second difference is the lack of a microprocessor or arbiter. The arbiter is no longer needed since the real-time scheduler makes sure that no two processes interfere with each other. The microprocessor is replaced by the custom hardware. The benefit of using hardware over a microprocessor program is speed. The program needs to fetch instructions and variables from memory, execute the instructions sequentially, and then store the results. The SMM hardware does not need to fetch and execute instructions, uses parallel processing to increase throughput, and collects the heap in a shorter amount of time. The only drawback to hardware is that it cannot execute as complex programs as software. That is one reason why Baker's copying collector is the chosen algorithm. It only requires a few pointers to execute.

Chapter 2 describes many of the reasons behind choosing Baker's copy collector for hardware implementation. The algorithm automatically compacts the heap and new object allocation is extremely quick. Baker's collection technique is also naturally suited for hardware acceleration. The entire heap is organized by only three pointers. These three pointers are easily represented in hardware by three registers. The location of these pointers automatically denote the color of objects, what object to scan next, what object to copy next, and where to allocate new objects. The copy collector is elegant in its simplicity. This is in contrast to the mark-sweep and reference counting algorithms.

The main reason for eliminating the mark-sweep algorithm from contention in Chapter 2 was because of the unbounded allocation time and the fragmentation of the

heap. Both of these reasons make mark-sweep unsuitable for a real-time system. However, the mark-sweep method is also awkward to implement in hardware. Mark-sweep was originally developed for software execution. The heap is traced using either a recursive algorithm or an auxiliary stack. Both methods are fine for software execution but are not optimal for hardware implementation. The recursive algorithm is impossible to implement in hardware. The auxiliary stack method may be done in hardware, but algorithms are needed to guard against overflow on top of the extra hardware that is necessary to manage the stack. These algorithms include the Boehm-Demers-Weiser that is discussed in Chapter 1. The reference counting algorithms do not collect cyclic data. Most reference counting collectors get around this obstacle by using a backup mark-sweep collector, but this solution also suffers from the problems that are mentioned earlier in the paragraph. All three classical garbage collection algorithms were originally developed in software but only the copy collector is easily translated into hardware.

The smart memory module communicates with the system through eight input/output lines. These I/O lines are shown below in Figure 3.4.

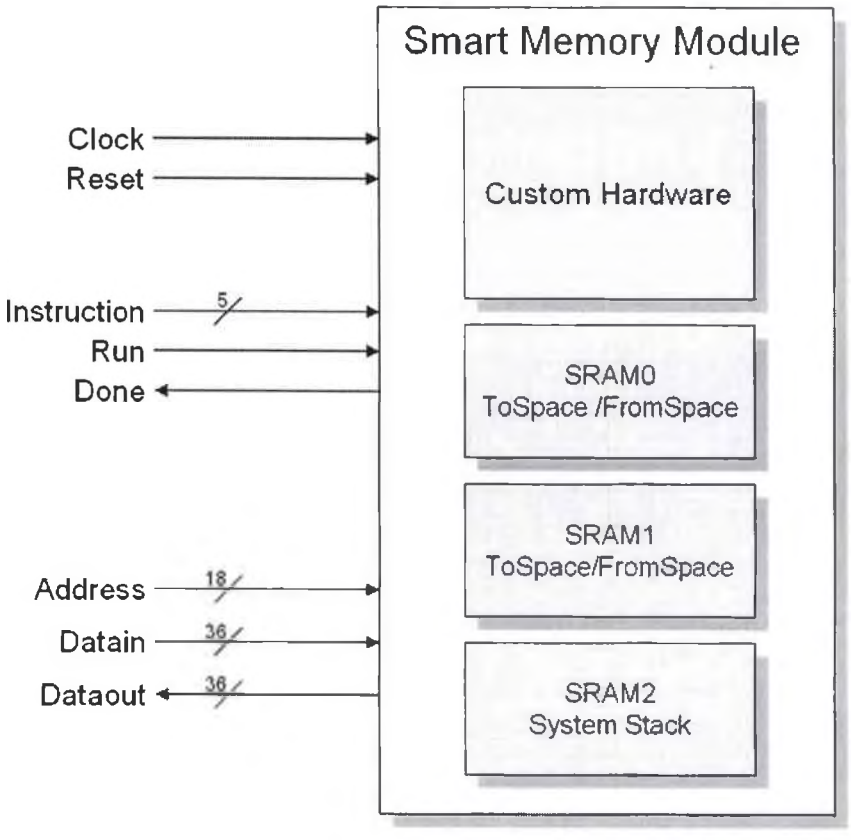

Figure 3.4: The Input/Output of the Smart Memory Module

Figure 3.4 shows the input and outputs of the smart memory module. The 'Clock' input is obviously the clock for the entire memory module. The 'Reset' line initializes all registers, timers, and pointers to their default values. An active low signal from the reset line supercedes all other input to the module. The 'Instruction' input is five bits wide and allows for a range of operations. Below is a table describing the smart memory module's basic instruction set.

Table 3.1: A List of the Smart Memory Module's Instructions

Instruction	Opcode	Description
Read	0	Reads a word of data from memory. The location of the word is specified on the 'Address' input line. Output is on the 'Dataout' line.
Write	1	Writes a word of data to memory. The location of the word is specified on the 'Address' input line. The incoming data is on the 'Datain' line.
New	2	Allocates space in memory for a new object. The size of the object is specified on the 'Datain' line.

		The address for the objects new location is given on the 'Dataout' line.
Garbage Collect	3	Initiates a collection period. The length of the collection time is preset by the GC timer.
Set GC Timer	4	Sets the number of clock cycles that are allowed for a collection period. The 'Datain' line contains the new value of the timer.
Set Free Space Size	5	Sets the minimum number of words between the New and Free pointer before of space switch occurs. The 'Datain' line contains the new value of the register.
PUSH Stack	6	Pushes a data word onto the system stack. The incoming word is on the 'Datain' line.
POP Stack	7	Pops a data word off the system stack. The output word is on the 'Dataout' line.

The instructions that are listed in the table above are executed when a valid instruction is inputted to the 'Instruction' line and the 'Run' line is asserted. The system is notified of the task completion when the 'Done' output line is asserted. Chapter 4, 5, and 6 explores the details of each instructions execution cycle and their state machines.

3.3 Implementation

The hardware implementation of the garbage collector is accomplished by using a hardware description language (HDL). A HDL is a high level programming language that is used to model the operation of a digital circuit. A compiler is then used to synthesize the HDL into a digital circuit layout. Digital design was previously accomplished by transistor layout at the schematic level. This process is slow, labor intensive, and costly. The development of HDLs in the early eighties represents a dramatic improvement in digital design. A single line of behavioral HDL code may represent dozens, if not hundreds, of transistors on the schematic layout. HDLs dramatically reduce the development time necessary for large digital devices. The result of a complied HDL design might be less efficient resource wise than a hand drawn schematic, but the savings made in design, testing and debugging time compensate for this short-coming.

There are several HDLs in industry today. The hardware description language that is used in this project is called Very High Speed Integrated Circuit (VHSIC) hardware description language (VHDL). This language was originally developed by the

Department of Defense as a way to unify its digital components into a single standard. VHDL became widespread during the late 1980s and was used on projects such as the F-22 [14]. The language is supposed to be self-documenting and provides convenient means for which to combine smaller modules into one large design.

The smart memory module is composed of five VHDL modules. Figure 3.5 shows the hierarchal structure of entities.

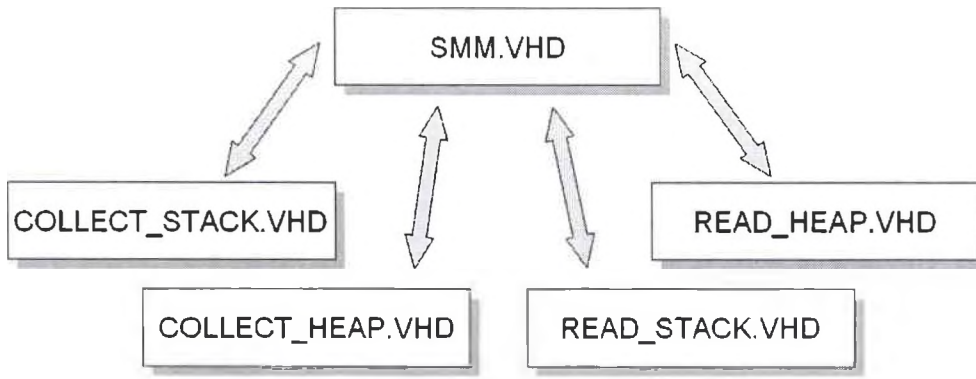

Figure 3.5: The Hierarchal View of the VHDL Entities

The high level VHDL module, smm.vhd, contains several registers and a state machine. The registers contain data including the Scan, Free, and New pointers along with timers that insure real-time execution. The state machine controls each instruction that the smart memory module receives. Simple instructions such as memory writes, timer controls, and debugging functions are executed directly in the high level VHDL module. More complex functions such as collecting the heap and the read barrier are handled by low level VHDL modules. The high level module enables these low level functions at the appropriate time.

The two models that control the collection process are collect_stack.vhd and collect_heap.vhd. Collect_stack.vhd scans the system stack for root references to the heap. Any object whose reference is found on the stack is automatically copied to the ToSpace. The collect_heap.vhd module starts after the entire system stack is scanned for references. This module scans the objects that are already copied to ToSpace for descendant. The descendants are copied to the ToSpace and the process repeats until no descendants are left.

The two remaining modules, read_stack.vhd and read_heap.vhd, are responsible for enforcing the read barrier. The read_stack.vhd is activated whenever the system tries to POP a word off the stack. A system stack is not normally protected by a read barrier in most garbage collection systems. A collector usually scans the entire stack for references during the first collection round. This setup is not acceptable for a hard real-time system because the length of the system stack is unbounded, and scanning the whole stack might take too long. Therefore, the system stack is scanned incrementally in this design. The one drawback is that it now must be protected by a read barrier. The read_heap.vhd module checks for read barrier violations when the system attempts to read memory from the heap.

The target device for the garbage collector that is written in VHDL is a field programmable gate array (FPGA). Traditional circuit designs are custom-made in a factory and can only be programmed once. A FPGA is capable of being programmed a near infinite number of times. This capability allows for easily upgrading and quickly debugging the development system. Current FPGA devices contain millions of transistors and are capable of speeds in the hundreds of megahertz [15].

The smart memory module prototype is built on a Xilinx Virtex FPGA. This FPGA contains 1,124,022 system gates and 27,648 logic cells. A logic cell consists of 4-input look-up table, one flip-flop, and carry logic [15]. The maximum clock speed of the chip is one hundred megahertz. The FPGA is connected to four memory blocks through four memory buses. Figure 3.6 shows the system setup.

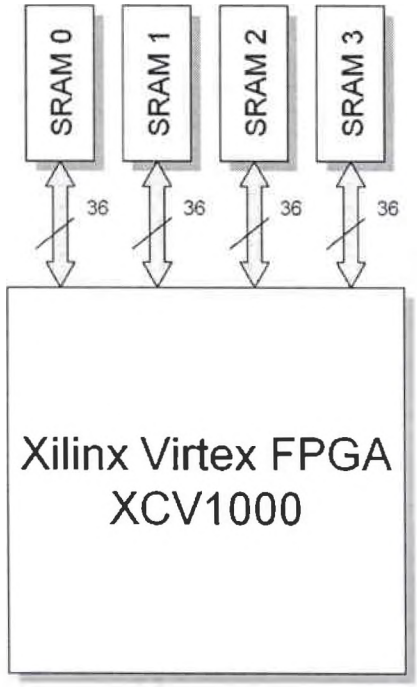

Figure 3.6: The Prototype Smart Memory Module [16, 17]

SRAM 0 and SRAM 1 are used as the ToSpace and FromSpace. SRAM 2 is used to store the system roots in a stack. SRAM 3 is not used in the prototype. All custom hardware runs on the FPGA. There are several important items to notice in Figure 3.6. The first is that each memory module has an independent bus. This architecture allows for parallel high speed copying between the ToSpace and FromSpace. A system with only one memory bus is not optimal for a copying garbage collector because the system cannot read from the FromSpace and write to the ToSpace at the same time.

The memory module and the memory bus have a word width of thirty-six bits. The first thirty-two are for the systems data. The remaining four bits are used by the collector to tag data. The data tags mark whether a memory word is plain data, a reference, the object's header information, or a forwarding pointer. This extra information aids the collector in scanning and sorting the heap. The memory manager would have to look up a class definition each time it scans an object if the data tags are not included. This process is time intensive and makes garbage collection too long for a real-time system. The data tags allow the collector to instantaneously identify the nature the data word it retrieves from memory.

3.4 Summary

The garbage collector is to be designed in a memory module. This packaging of the module allows for minimal delays between the memory manager and memory cells. This design characteristic is similar to that of Kelvin Nilsen's GCMM unit. The main difference between the two projects is that Nielson utilizes a general purpose processor to execute his collection algorithm. This research use embedded hardware to perform Baker's copying algorithm. The embedded hardware is to theoretically run faster than a general purpose processor and therefore be better suited for hard real-time systems.

The unit is built on a Xilinx FPGA that is linked to three independent memory devices. The parallel bus structure allows for optimal copying between the ToSpace and FromSpace. The memory words in each SRAM module are thirty-six bits wide to accommodate data tags. The data tags allow the collector to rapidly analyze the content of memory words.

The next three chapters concentrate on the inner workings of the VHDL modules that are described above. Chapter 4 focuses on the high level smm.vhd module. Chapter 5 looks at the garbage collectors in collect_stack.vhd and collect_heap.vhd. Finally, Chapter 6 examines the read barrier that is described in read_stack.vhd and read_heap.vhd.

Chapter 4

Implementation of the Smart Memory Module

Chapters 1 and 2 discuss the algorithm selection for a real-time garbage collecting system. Chapter 3 focuses on the design and implementation of smart memory module. The purpose of this chapter is to explore the inner workings of the smart memory module. A general description on what a state machine is and how it aids the smart memory module design is given. The state machines of the five VHDL modules are examined in detail along with information on how the modules communicate with other modules. In addition, the appendix contains low-level register transfer language (RTL) tables that describe the entire smart memory module functions listed below.

4.1 The State Machine

One problem that is encountered while programming in hardware description language (HDL) is that the code is not sequential like traditional software programming. Languages such as C and Java execute their code one line after another in a sequential format. Hardware description languages are not sequential. They execute every line at once. This problem arises because the HDL is programming actual hardware devices that run continuously. Programming a device such as the smart memory module asynchronously is impracticable.

The solution to the non-sequential problem in HDL is to construct a state machine. The state machine sets conditions on the hardware so that only certain sections execute at a given time. The state machine allows the HDL program to be broken into manageable parts. Each state in the state machine executes a part of the program, and all the states working together produce a smart memory module.

4.2 Mid-Level Design

The smart memory module is centered on a state machine. This machine controls how the module responds to different types of input. Figure 4.1 shows a mid-level abstraction of the state machine for the garbage collecting memory module.

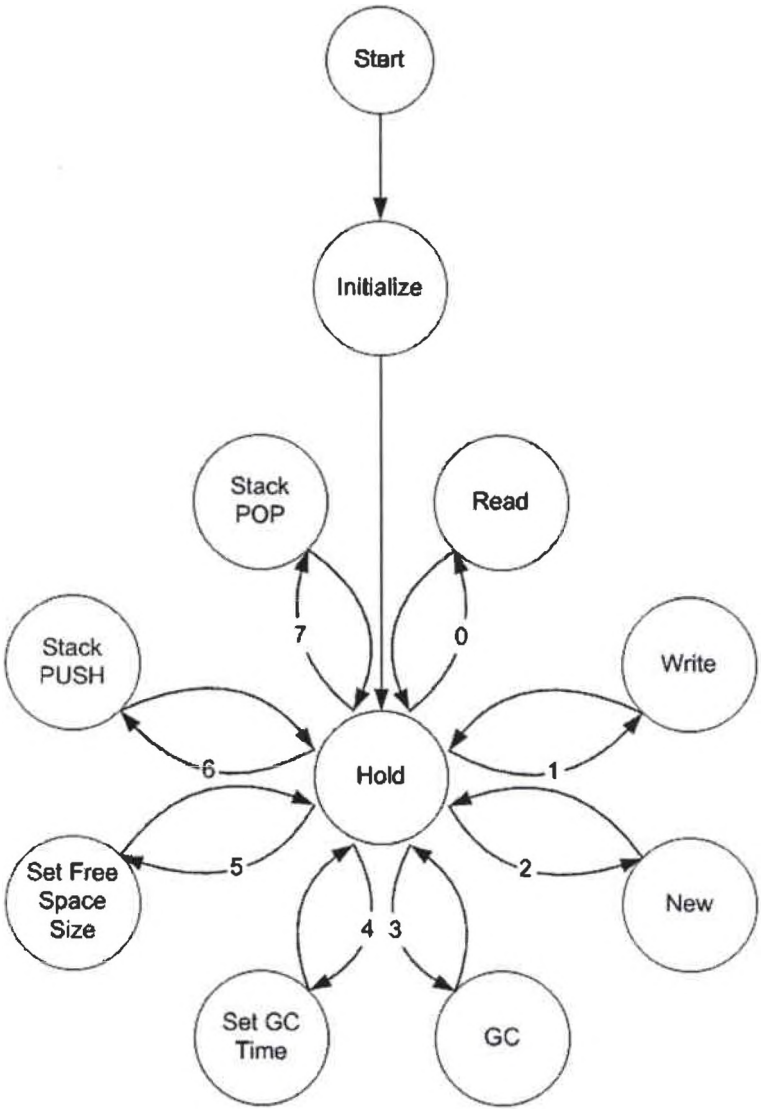

Figure 4.1: The State Machine for the Smart Memory Module

Figure 4.1 is referred to as a mid-level overview of the state machine because some miscellaneous states, such as wait states, are left out to clarify the algorithm flow. A full register transfer language (RTL) description of the entire garbage collection system is available in the appendix.

The smart memory module always begins in the 'Initialize' state after either a reset or start-up. The state resets all internal pointers and timers. The system then immediately proceeds to the 'Hold' state. The SMM stays in the 'Hold' state until a valid instruction is given to the system. The potential states that the system may move to from 'Hold' are shown above in Figure 4.1 along with the valid operational code. The remainder of the chapter discusses the state machines surrounding the eight SMM instructions. The 'Write,' 'New,' 'Set GC Time,' 'Set Free Space Size,' and 'Stack PUSH' commands are relatively simple to execute. Their state machines are in the smm.vhd module. The 'Read,' 'GC,' and 'Stack POP' commands are more complicated instructions. Their state machines are located in the four low-level VHDL entities. These modules are collect_stack.vhd, collect_heap.vhd, read_stack.vhd, and read_heap.vhd.

4.2.1 The 'Write' Command

The system uses the 'Write' instruction when it wishes to record data onto the heap. The system specifies the desired address on the 'Address' line and the data on the 'Datain' line. The state machine flow for the 'Write' function is displayed below.

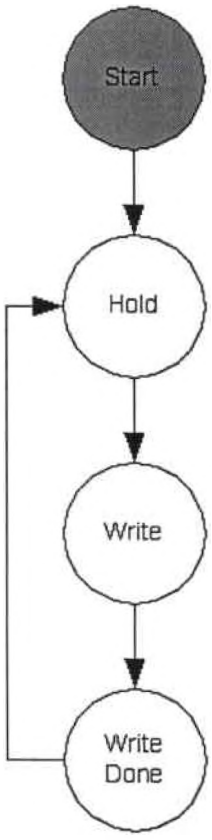

Figure 4.2: The Write Instruction

The SMM stays in the 'Hold' state until a valid 'Write' instruction is received. The state machine then progresses to the 'Write' state. The data is written to the denoted address during this state. The SMM then proceeds to the 'Write Done' state. This state notifies the system that the 'Write' command is completed by asserting the 'Done' line. The state machine then progress to the 'Hold' state to await the next instruction from the system.

4.2.2 The 'New' Command

The system issues a new command when it needs to allocate a new object on the heap. The system specifies the size of the new object on the 'Datain' line. The state machine for the 'New' instruction is shown below.

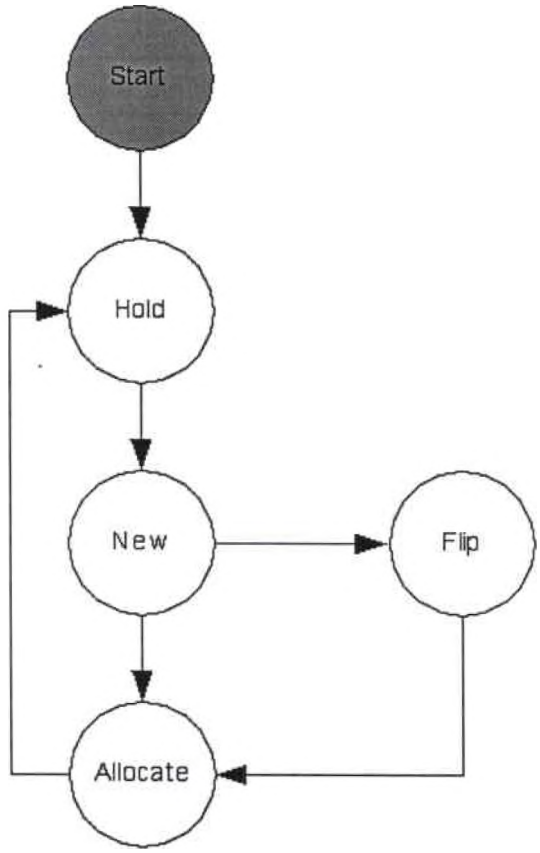

Figure 4.3: The New Command

The SMM stays in the 'Hold' state until a valid 'New' instruction is received. The SMM then proceeds to the 'New' state. The dynamic memory module checks to see whether there is enough free memory space remaining to allocate the new object. The SMM precedes to either the 'Flip' space if there is insufficient space or to the 'Allocate' state if there is enough memory space for the new object. The 'Flip' state is accessed when there is not adequate room left in the ToSpace. The state flips to ToSpace and FromSpace titles, resets the New pointer to the end of the ToSpace, and notifies the garbage collector that a new collection cycle has begun. The 'Allocate' state uses the New pointer and the new object's size to calculate and assign a location for the data. The state then outputs the memory location for the new object on the 'Dataout' line and asserts the 'Done' line.

4.2.3 The 'PUSH' Command

The PUSH command is for putting data onto the system stack that is located in the smart memory module. The system inputs a valid 'PUSH' command on the instruction line and data on the 'DataIn' line. Below is the state diagram for the PUSH command.

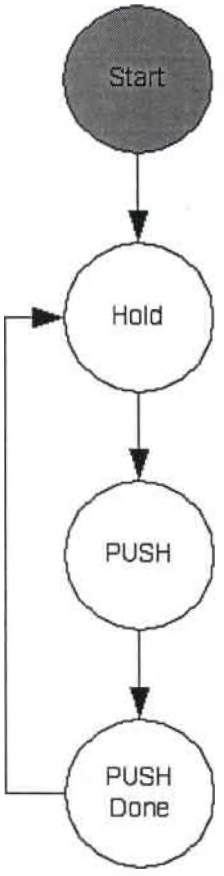

Figure 4.4: The Push Instruction

The SMM stays in the 'Hold' state until a valid 'PUSH' instruction is received. The module then proceeds to the 'PUSH' state. This state puts the new word of data on top of the system stack. The 'PUSH Done' state is executed next. This state notifies the system of task completion by asserting the 'Done' output line.

4.2.4 The 'Set GC Time' Command

A real-time system has the ability to change the length of collection time by using the 'Set GC Time' instruction. This instruction allows a real-time program to fit garbage collection cycles into various time slots. The system changes the garbage collection time by sending a valid 'Set GC Time' command on the instruction line and the new time length, in clock cycles, on the 'DataIn' line. Below is a diagram of the state machine for the 'Set GC Time' instruction.

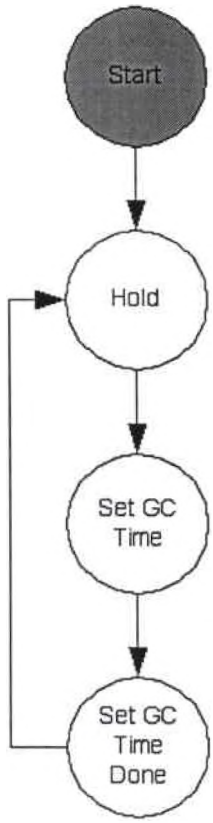

Figure 4.5: The 'Set GC Time' Instruction

The SMM stays in the 'Hold' state until a valid 'PUSH' instruction is received. The hardware then moves to the 'Set GC Time' state. The value of the garbage collector's timer register is modified during this state. The memory module then progresses to the 'Set GC Time Done' state. This state notifies the system of task completion by asserting the 'Done' output line.

4.2.5 The 'Set Free Space Size' Command

The 'Set Free Space Size' instruction allows a real-time system user to specify the amount a free space the ToSpace should always retain. This value can range from zero to the size of the entire heap. The system changes the garbage collection time by sending a valid 'Set Free Space Size' command on the instruction line and the new free space length, in memory words, on the 'DataIn' line.

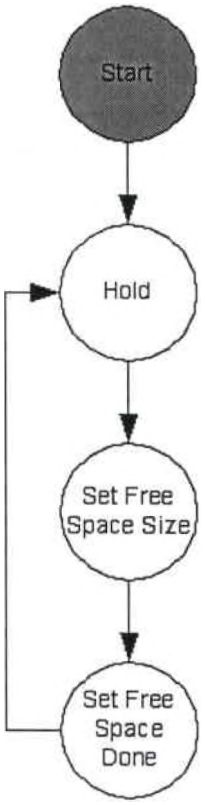

Figure 4.6: The 'Set Free Space Size' Command

The SMM stays in the 'Hold' state until a valid 'PUSH' instruction is received. The hardware then moves to the 'Set Free Space Size' state. The value of the garbage collector's Free Space register is modified during this state. The memory module then progresses to the 'Set Free Space Size Done' state. This state notifies the system of task completion by asserting the 'Done' output line.

4.3 The Garbage Collector

The more complicated function of garbage collection is left to two low-level VHDL entities. These entities are shown in Figure 3.5 of Chapter 3. They are collect_stack.vhd and collect_heap.vhd. The state machines for these functions are much more complicated than the previous five instructions' state machines. The collection of the heap is done in two phases. The first phase is scanning the system stack for any root references. Any object that is directly referenced by the root is copied to the ToSpace. This phase of collection is handled by collect_stack.vhd. The second phase of collection

is scanning the ToSpace for references to objects in the FromSpace. This phase takes place after the objects that are directly referenced by the system stack have been copied. The second phase copies every live object still in FromSpace to ToSpace. The collect_heap.vhd entity is responsible for the execution of phase two.

The high level smm.vhd entity still controls the execution of the smart memory module. It initiates an incremental collection cycle when the system sends the 'GC' command on the 'Instruction' input line. The high level module's state machine for an incremental garbage collection cycle is shown below.

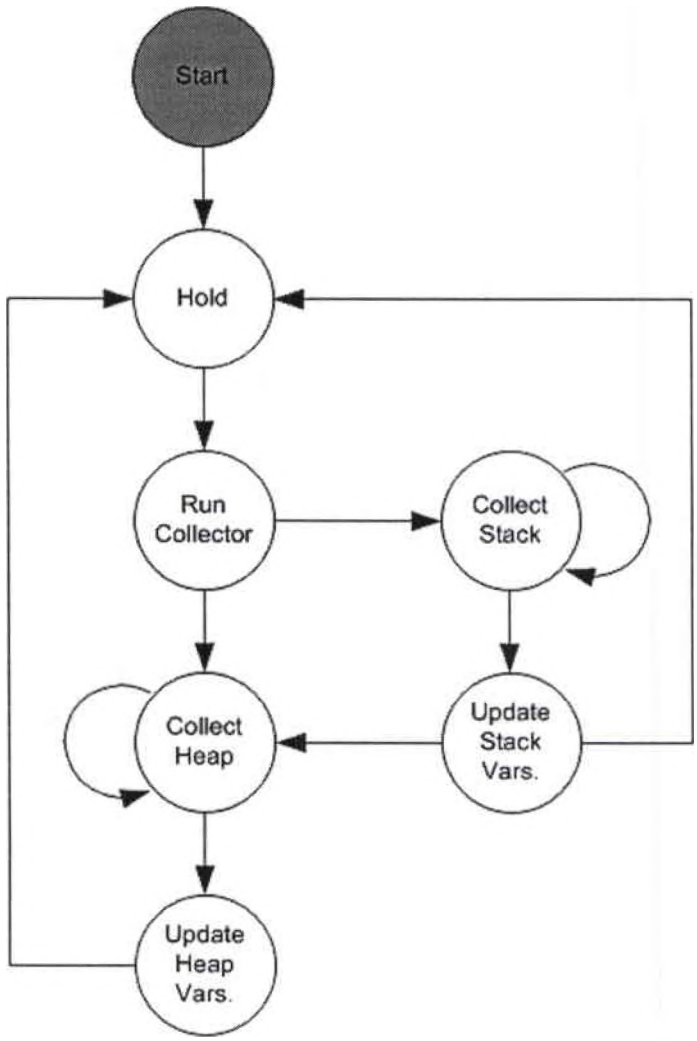

Figure 4.7: The High-Level State Machine for Garbage Collection

The state machine that is shown in Figure 4.7 executes in the smm.vhd module. The collector stays in the 'Hold' state until it receives the 'GC' command. The 'GC' command begins an incremental garbage collection cycle. The machine then proceeds to the 'Run Collector' state. The 'Run Collector' state determines whether the system stack needs to be scanned for root references or if the collector should scan the heap for descendent objects that are still in FromSpace. The machine proceeds to the 'Collect Stack' state if there are still portions of the system stack that have yet to be visited by the collector. The state machine only moves to the 'Collect Heap' state if the entire stack is scanned and the collector needs to check for root object descendents still in FromSpace.

4.3.1 The Stack Collector

The smm.vhd module activates the collect_stack.vhd module when the state machine that is shown in Figure 4.7 is in the 'Collect Stack' state. The smm.vhd module maintains its current state as the collect_stack.vhd module executes its state machine. The interface between the smm.vhd and collect_stack.vhd modules is shown below.

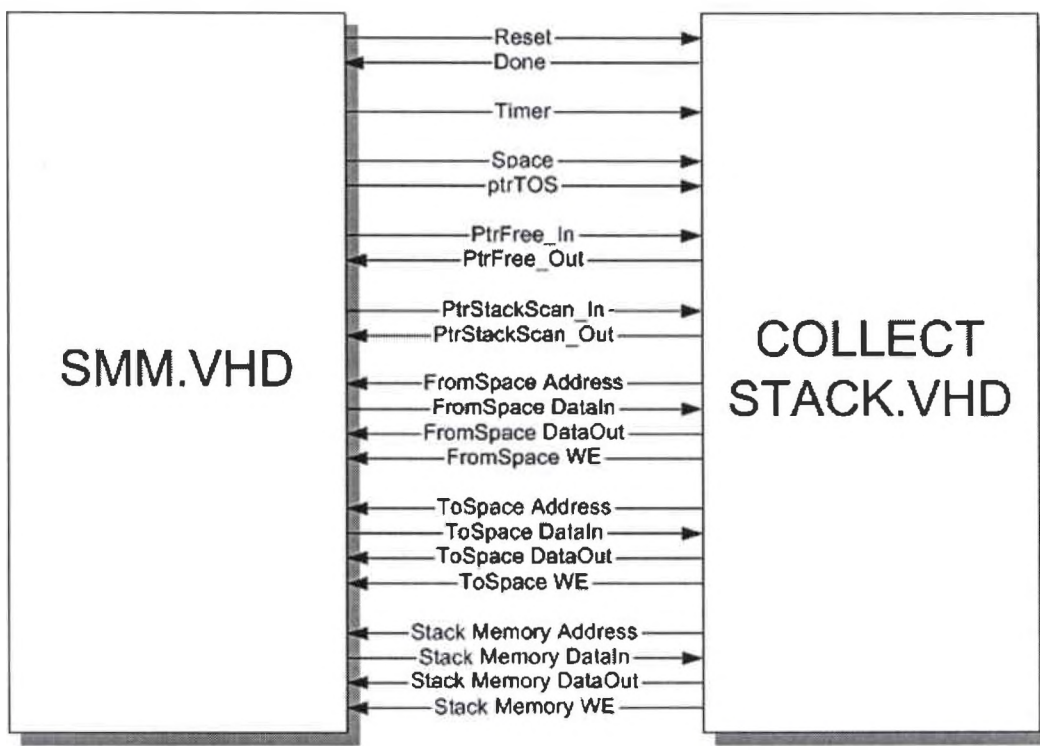

Figure 4.8: The Interface between SMM.VHD and COLLECT_STACK.VHD

The high level module starts the collect_stack.vhd module by setting the 'Reset' line to active low. The stack collector entity informs the smart memory module of its task completion by asserting the 'Done' line. The 'Timer' line from the SMM informs the collector of how much time remains during the collection cycle. The collector is allowed to read the Space, Top of Stack, Free, and Stack Scan registers. It is also allowed to write to the Free and Stack Scan registers. The stack collector entity is allowed to read and write to all three memory modules.

The collect_stack.vhd module initiates its state machine when its 'Reset' line is set to active low. Below is the state machine for the collect_stack.vhd module.

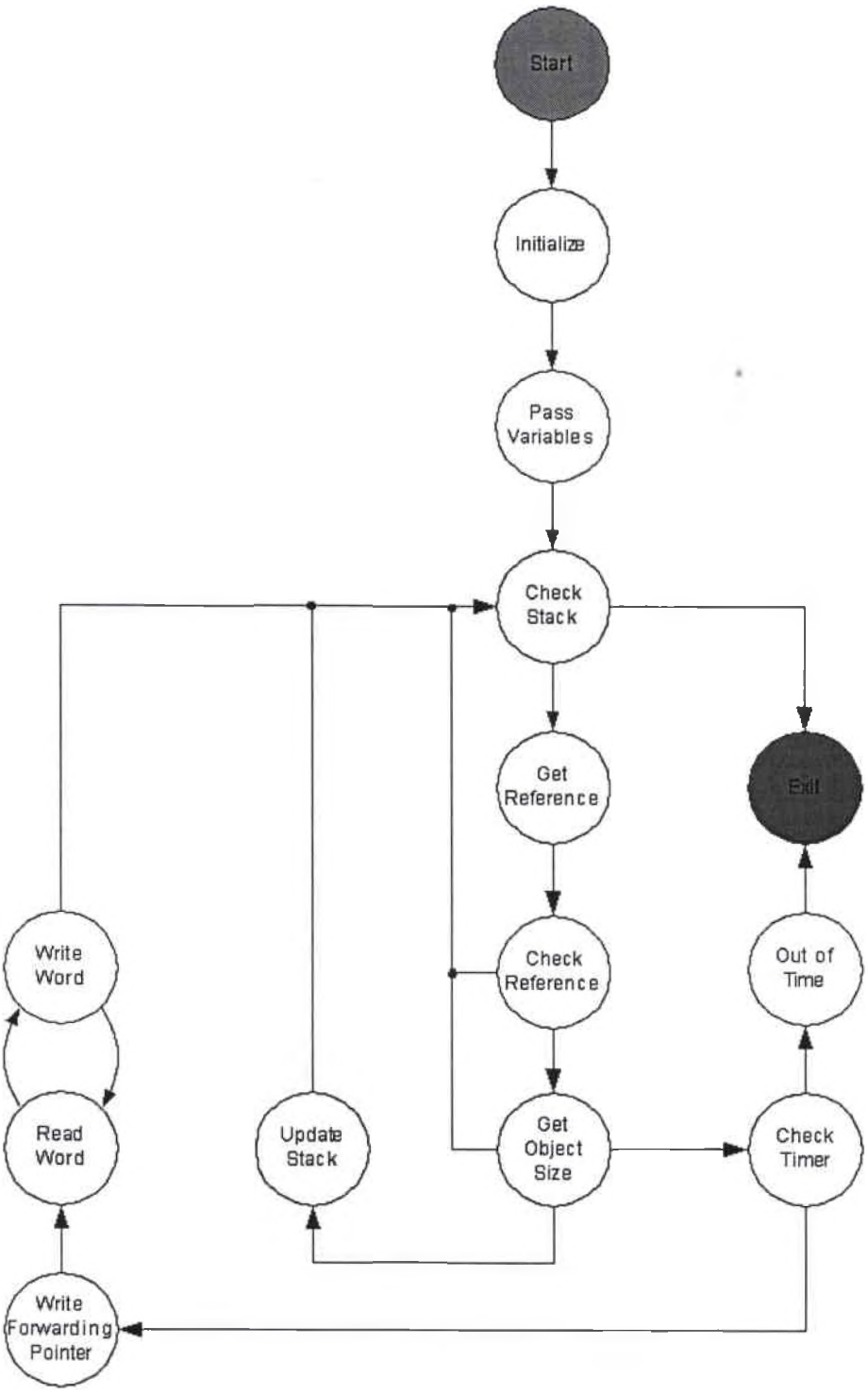

Figure 4.9: The Garbage Collector State Machine for the System Stack

Figure 4.9 shows that the state machine for scanning the stack is much more complex than the other instructions that are shown in the sections above. The purpose of this module is to scan the system stack, and identify references to objects in the heap.

The hardware starts to scan the stack from the stack's bottom and gradually advances to the top of the stack. The bottom of the stack contains the most stable data. Data at the top of the stack is often removed by the system during program execution. Collecting from the top of the stack initially is dangerous since objects may be copied that quickly die. Starting from the bottom of the stack allows for those short-lived objects to die before they are collected.

The collect_stack.vhd module starts in the 'Initialize' state. This state sets all internal registers to their default values. The machine then progresses to the 'Pass Variables' state. Variables that are local to the smm.vhd entity are passed to the collect_stack.vhd module during this period. These variables include the current location of the Free pointer and the StackScan pointer. The StackScan pointer identifies the parts of the stack that have already been scanned by the collector. The memory manager advances to the 'Check Stack' state after all the variables have been transferred.

The 'Check Stack' state performs two checks that determine the next state. The first check tests whether the module is finished scanning the stack for references. This condition is true if the StackScan pointer is equal to the top of the stack. The second test is to check the garbage collection timer. The collector needs eleven clock cycles in order to advance and still meet its real-time requirements. The entity proceeds to the 'Exit' state if either the entire stack is scanned or if there is not enough time. The collector goes to the 'Get Reference' state if both these conditions are not true.

The 'Get Reference' state retrieves the data word from the system stack whose address is currently in the StackScan pointer. The StackScan pointer is incremented to point to the next highest object on the stack. The garbage collector then analyses the stack data in the 'Check Reference' state. The collector moves to the 'Get Object Size' state if the stack data is a reference to FromSpace. The module moves to the 'Check Stack' state if the data is not a reference to FromSpace.

The 'Get Object Size' state retrieves the object header from the referenced object in FromSpace. The object header tells the collector whether the object still needs to be collected or if the object is already in ToSpace and the system stack's reference just needs to be updated to the new location. The module moves to the 'Update Stack' state if the object's header does contain a forwarding pointer. The forwarding pointer contains

the objects new location in ToSpace. The system stack's reference is updated to the new ToSpace location. The module then returns to the 'Check Pointers' state. The state machine advances from the 'Get Object Size' state to the 'Check Timer' state if the object's header does not contain a forwarding pointer.

The 'Check Timer' state analyses whether the garbage collector has enough time to copy the object. The maximum time allowed for a garbage collection cycle is specified by the system, and the collector cannot go over this time. The current state knows how large the object to be copied is and how long it takes to copy. The collector compares this data with the time remaining in the collection cycle. The collector goes to the 'Out of Time' state if there is not enough time. This state decrements the StackScan pointer so that the collector begins at the current object at the next collection cycle.

The garbage collector goes to the 'Write Forwarding Pointer' state if there is enough time to copy the object. This state updates the system stack with the object's new location, writes a forwarding pointer at the object's old location, and writes the object's new header information in ToSpace. The object is copied to the location that is specified by the Free pointer. The module then begins a read and write loop in the 'Read Word' and 'Write Word' states. The object's data is read from FromSpace during the 'Read Word' state and written to the ToSpace during the 'Write Word' state. This process repeats until the entire object is copied. The Free pointer is updated to the new free location and the module returns to the 'Check Stack' state.

4.3.2 The Heap Collector

The collect_stack.vhd entity returns control to the smm.vhd state machine that is shown in Figure 4.7 when it reaches the 'Exit' state. The smm.vhd state machine advances to the 'Update Stack Variables' state. This state informs the smm.vhd module of any changes done to the Free and StackScan pointers. It then returns to the 'Hold' state if there is not enough time to proceed with collection. The garbage collector advances to heap collection if time permits. The smm.vhd module activates the collect_heap.vhd module when in the 'Collect Heap' state. The purpose of this module is to search the heap for descendents of the root objects still in the FromSpace. These descendents are copied over to the ToSpace, and then searched for their own descendents.

This process repeats itself until the Free and Scan pointers are equal. Below is a diagram showing the interface between the smm.vhd and collect_heap.vhd modules.

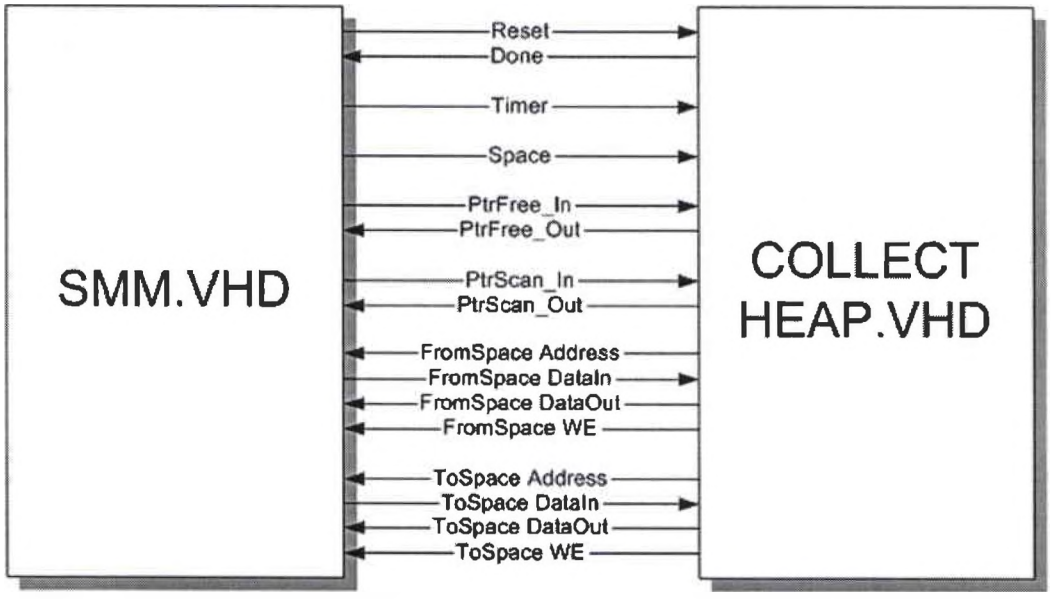

Figure 4.10: The SMM.VHD and COLLECT_HEAP.VHD Interfaces

The heap collection module is activated by setting its 'Reset' line to active low. The collector informs the SMM of task completion by asserting the 'Done' line. The heap collector maintains its real-time limits by monitoring the 'Timer' input line. The module has read access to the 'Space,' 'PtrFree,' and 'PtrScan' registers in smm.vhd entity. The collector is also allowed to write to the Free and Scan pointers in smm.vhd. The collect_heap.vhd module is permitted read and write access to only the ToSpace and FromSpace. The state machine for the collect_heap.vhd entity is shown below.

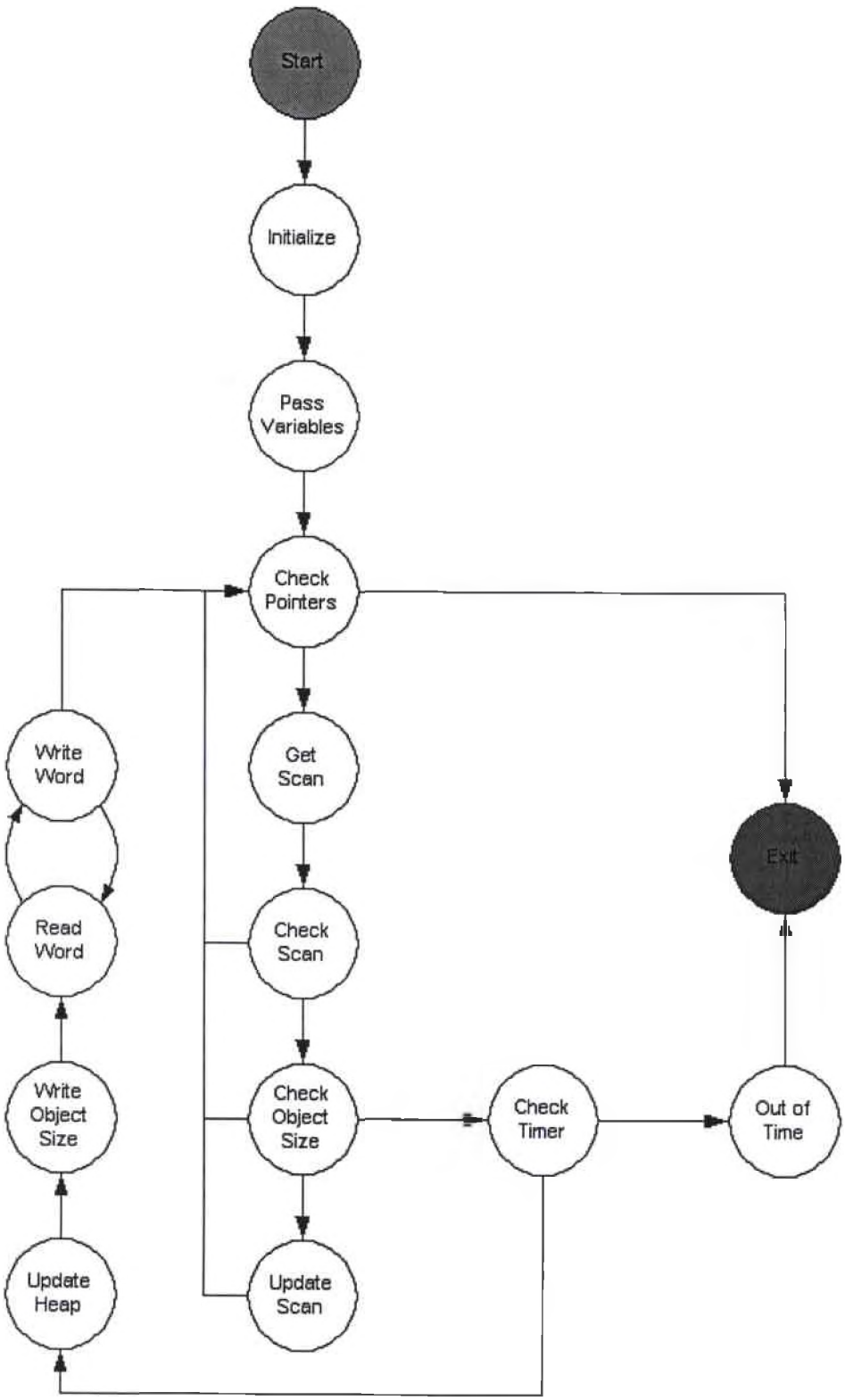

Figure 4.11: The Garbage Collector State Machine for the Heap

The smm.vhd holds the state of 'Collect Heap' until the state machine that is shown above reaches the 'Exit' state. The heap collector begins in the 'Initialize' state. This state sets all registers to their default levels. The collector proceeds to the 'Pass Variables' state. Variables that are local to the smm.vhd entity are passed to the collect_heap.vhd module during this period. These variables are the Free and Scan pointers. The state machine advances to the 'Check Pointers' state after the variables are transferred. This state looks for two conditions that might cause the collector to exit the collect_heap.vhd module. The first condition is if the Free and Scan pointers are equal. The second condition is if there is less than eleven clock cycles remaining in the collection cycle. These clock cycles are the minimum number needed to maintain the real-time limits. The garbage collector moves to the 'Get Scan' state if both of these conditions are false.

The 'Get Scan' scan state retrieves the ToSpace memory that is indicated by the Scan pointer. The collector then checks the memory word in the 'Check Scan' state. The state machine returns to the 'Check Pointers' state if the fetched word is not a reference to an object in FromSpace. The garbage collector moves to the 'Check Object Size' state if the retrieved word is a reference to FromSpace. The 'Check Object Size' state gets the header of the referenced FromSpace object. The object header tells the collector whether the object still needs to be collected or if the object is already in ToSpace and the heap's reference just needs to be updated to the new location. The module moves to the 'Updated Scan' state if the object's header contains a forwarding pointer to the ToSpace. The 'Update Scan' state writes over the FromSpace reference with the object's new ToSpace location. The collector then returns to the 'Check Pointers' state. The 'Check Object Size' state advances to the 'Check Timer' state if the header information contains the object's size in memory.

The 'Check Timer' state analyses whether the garbage collector has enough time to copy the selected object. The maximum time allowed for a garbage collection cycle is specified by the system, and the collector cannot go over this time. The current state knows how large the object to be copied is and how long it takes to copy that object. The collector compares this data with the time remaining in the collection cycle. The collector goes to the 'Out of Time' state if there is not enough time. This state

decrements the Scan pointer so that the collector begins at the current object at the next collection cycle

The collector goes to the 'Update Heap' state if there is enough time to copy the object. The state writes the new ToSpace location of the object to the original reference that began this collection process. The garbage collector then moves to the 'Write Object Size' state. This state writes the object's header information to its new ToSpace location and leaves a forwarding pointer in the FromSpace. The module then begins a read and write loop in the 'Read Word' and 'Write Word' states. The object's data is read from FromSpace during the 'Read Word' state and written to the ToSpace during the 'Write Word' state. This process repeats until the entire object is copied. The Free pointer is updated to the new free location and the module returns to the 'Check Pointers' state.

The heap collector returns control to the smm.vhd state machine that is shown in Figure 4.7 when it reaches the 'Exit' state. The smm.vhd state machine then moves to the 'Update Heap Variables' state. The values of the Scan and Free pointers are modified to reflect the changes that occurred in the collect_heap.vhd entity. The garbage collector then returns to the hold state to await the next system instruction.

4.4 The Read Barrier

The smart memory module uses a read barrier to protect the heap's data structure during incremental garbage collection. The read barrier prevents the mutator from catastrophically changing the heap's structure between garbage collection cycles. The barrier also protects against having multiple locations for one object. This particular problem occurs in any collector that moves data. The smart memory module uses two entities to enforce the read barrier. These modules are read_stack.vhd and read_heap.vhd. The first module enforces the read barrier on the system stack whenever the mutator tries to POP data. The second module implements the barrier when the system tries to read heap data.

4.4.1 Stack Read Barrier

The execution of the stack read barrier in the smm.vhd module is shown below.

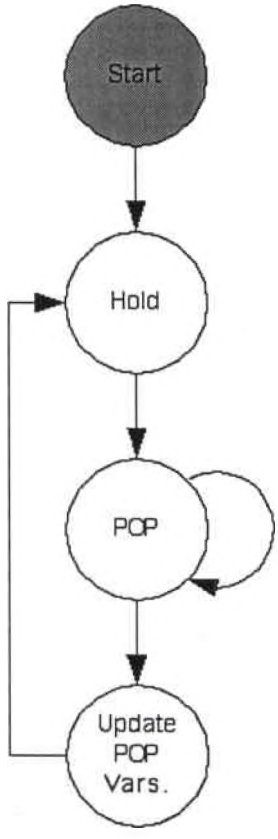

Figure 4.12: The Stack Read Barrier in SMM.VHD

The smm.vhd module maintains the 'Hold' state until a POP command is received by the smart memory module. The state machine then advances to the 'POP' state. This state activates the read_stack.vhd module. The smm.vhd state machine idles in the 'POP' state until the read barrier relinquishes control. The interface between the two VHDL modules is shown below.

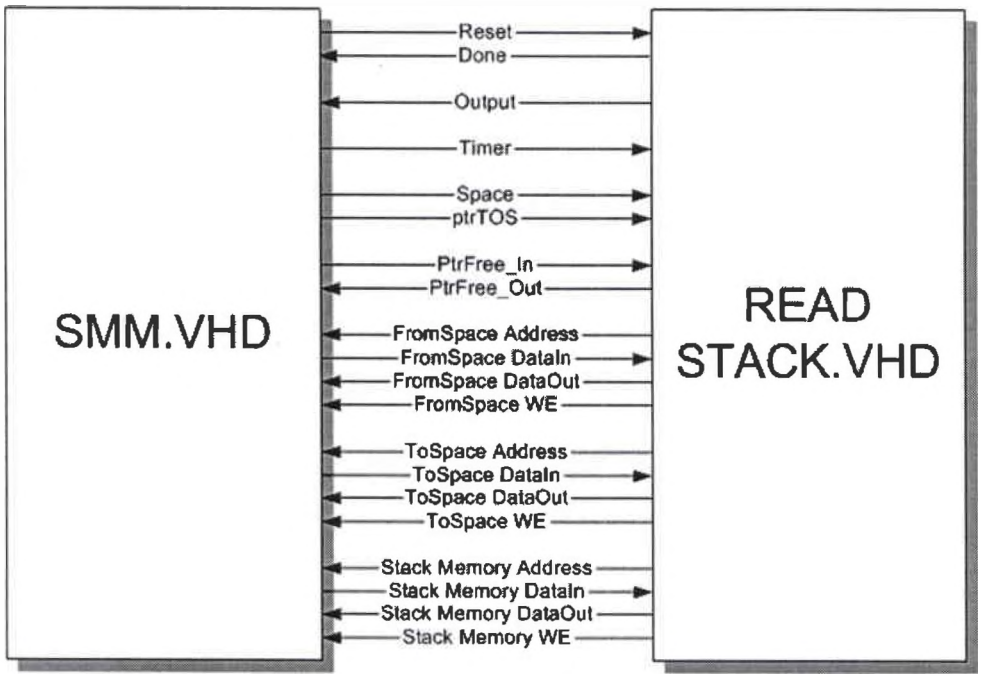

Figure 4.13: The SMM.VHD and READ_STACK.VHD Interface

The smart memory module starts the read barrier by putting the 'Reset' line to active low. The read barrier informs the system of completion by asserting the 'Done' line. The stack read barrier has read access to the 'Space,' 'ptrTOS,' and 'PtrFree' registers in smm.vhd. Furthermore, the barrier has write privileges with the Free pointer. The read_stack.vhd can read and write to all three memory modules. The output of the POP command is given on the 'Output' line. Below is a state diagram of the read_stack.vhd entity.

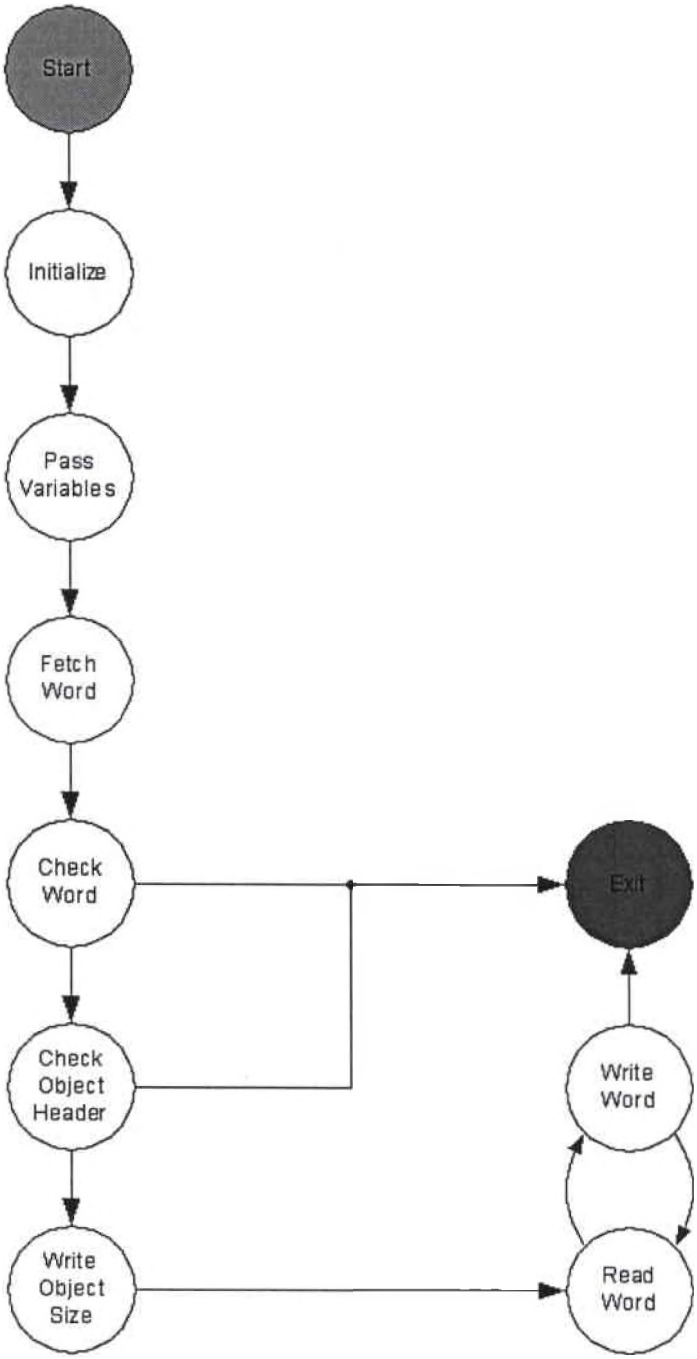

Figure 4.14: The Read Barrier for the System Stack

The module begins in the 'Initialize' state. This state sets all internal registers to their default values. The machine then progresses to the 'Pass Variables' state. The stack read barrier receives the current value of the Free pointer during this state. The entity then moves to the 'Fetch Word' state. This state retrieves the memory word off the

system stack that the mutator has requested. The module then advances to the 'Check Word' state. The 'Check Word' state checks the stack data word to see if it is a reference to an object in FromSpace. The read barrier exits the read_stack.vhd module if the word is not a reference to FromSpace. The output of the module at this time is the current data on the stack. The entity moves to the 'Check Object Header' state if the data is a FromSpace reference. This state examines the header of the referenced object. The read barrier moves to the 'Exit' state if the header information contains a forwarding pointer. This pointer indicates that the referenced object is already collected in ToSpace. The read barrier module outputs the current location of the object in ToSpace to the mutator.

The read barrier entity proceeds to the 'Write Object Size' state if the object header shows that the referenced object is in FromSpace. This state writes the FromSpace object's header to ToSpace at the Free pointer location. The module then begins a read and write loop in the 'Read Word' and 'Write Word' states. The object's data is read from FromSpace during the 'Read Word' state and written to the ToSpace during the 'Write Word' state. This process repeats until the entire object is copied. The Free pointer is updated to the new free location and the module returns control to smm.vhd. The output of the entity at this time is the new ToSpace location of the object.

4.4.2 Heap Read Barrier

The heap read barrier is active whenever the mutator tries to access data on the heap. The barrier analyzes the requested word to see if it is a reference to an object in FromSpace. The FromSpace object is evacuated to ToSpace if the heap word is a reference to it. The smm.vhd state machine for the heap read barrier is shown below.

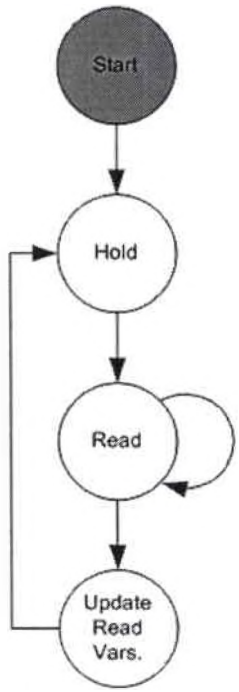

Figure 4.15: The SMM.VHD State Machine for the Heap Read Barrier

The smart memory module moves to the 'Read' state when it receives the valid read instruction. The smm.vhd module then activates the heap read barrier and maintains its present state until the barrier is finished. The interface between the smm.vhd and read_heap.vhd modules is shown below.

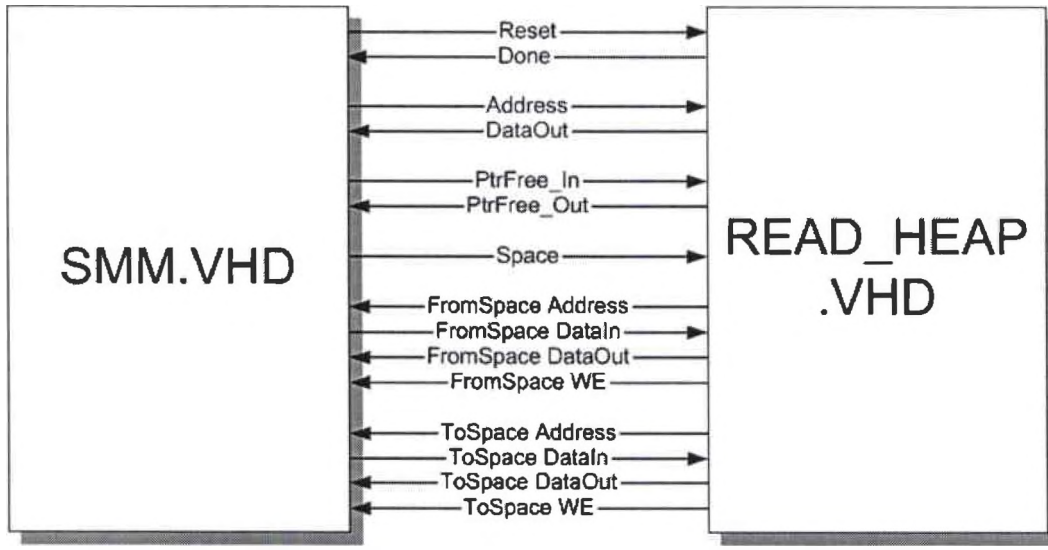

Figure 4.16: The SMM.VHD and READ_HEAP.VHD Interface

The high level module, smm.vhd, activates the heap read barrier by setting the 'Reset' line to low. The read barrier indicates task completion by asserting the 'Done' line. The requested heap address is on the 'Address' line and the resulting data is returned on the 'DataOut' line. References to the FromSpace are never returned by the read barrier. The barrier evacuates the FromSpace object to the ToSpace and returns the new reference. The read barrier also reads and writes to the PtrFree register on the appropriate interface lines. The read barrier is has access to both the ToSpace and FromSpace memory modules.

The heap read barrier begins execution when its reset signal is set to active low by the smm.vhd module. The read_heap.vhd module then begins its state machine execution. The state machine for the heap read barrier is shown below.

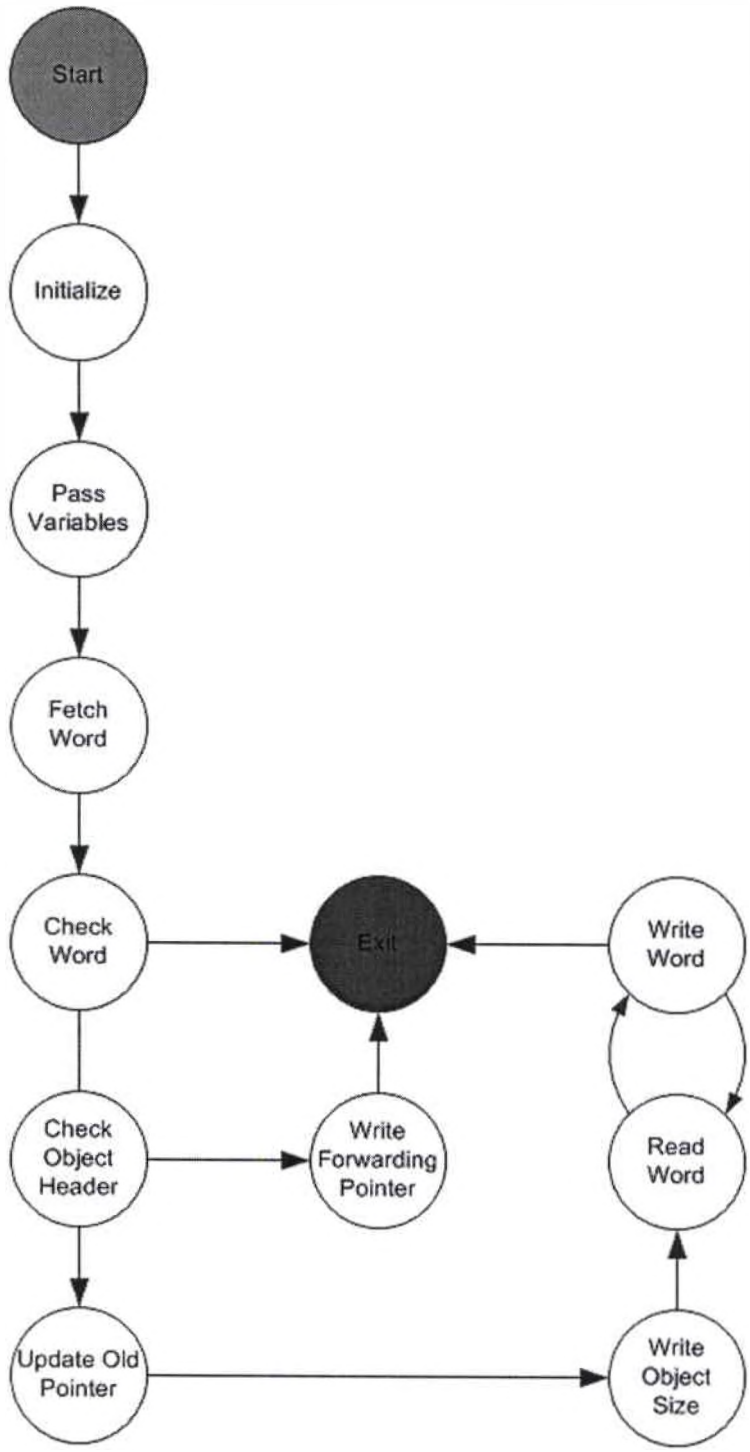

Figure 4.17: The Read Barrier for the Heap

The state machine begins in the 'Initialize' state. This state sets all internal registers to their default levels. The module then progresses to the 'Pass Variables' state.

This state loads the Free pointer's current value into the read_heap.vhd entity. The machine then advances to the 'Fetch Word' state. The 'Fetch Word' state retrieves the memory word that is specified on the 'Address' line from the smart memory module. The module then moves to the 'Check Word' state. The state checks to see whether the requested data is a reference to FromSpace or not. The read barrier exits if the memory word is not a FromSpace reference, and the output of the system is that memory word. The module progresses to the 'Check Object Header' state if the memory data contains a reference to FromSpace. The 'Check Object Header' state fetches the header information of the FromSpace Object and checks to see whether it is a valid object or a forwarding pointer. A forwarding pointer indicates that the object is already in ToSpace and the original reference is pointing to an old location.

The read_heap.vhd entity moves to the 'Write Forwarding Pointer' state if the object header contains a forwarding pointer. This state updates the original reference to the object's new location, and then exits the read barrier's module. The read barrier proceeds to the 'Update Old Pointer' from the 'Check Object Header' if the object header indicates that the object is still in FromSpace. The 'Update Old Pointer' state assigns the object a new location in ToSpace, writes a forwarding pointer to the object header in FromSpace, and updates the original reference with the object's new location. The 'Write Object Size' state records the object's new header information in ToSpace. The module then begins a read and write loop in the 'Read Word' and 'Write Word' states. The object's data is read from FromSpace during the 'Read Word' state and written to the ToSpace during the 'Write Word' state. This process repeats until the entire object is copied. The Free pointer is updated to the new free location and the module returns control to smm.vhd. The modules output is a reference to the object's new location in ToSpace.

Chapter 5

Design and Simulation Results

The topic of this chapter is to show the smart memory module's functionality and timing results. The first portion of the chapter deals with environment that is used to verify the custom hardware's functional specifications. A brief example is given for one test heap. The text then moves to the timing results of the smart memory module after the system is verified to work properly. Finally, the some of results of the SMM are compared to Kelvin Nilsen's GC memory module.

5.1 Verification of the Smart Memory Module

The main result of the smart memory module is whether the system works as designed. There is little research on implementing a garbage collector using just a custom embedded logic. In this respect, the SMM is a proof of concept device that demonstrates the feasibility of using embedded hardware for a real-time garbage collector. The current incarnation of the smart memory module utilizes a simple but effective garbage collection algorithm. The success of this version opens the door to implementing more sophisticated collection methods in embedded hardware devices. The SMM is built using the Synplify design environment suite, and the system's design is verified by using the Modelsim hardware simulator. Modelsim is a state-of-the-art HDL simulator that is widely recognized as industry standard. The development board that houses the SMM prototype comes with HDL testbench files. These files allow Modelsim to accurately test both the functional and timing aspects of a hardware design. This type of accuracy allows for a convenient way to test the functionality of the smart memory module. The Modelsim environment accurately portrays how the SMM executes on hardware and allows for the designer to view all internal registers in the module. Both

of these features combined provide an excellent means in which to validate the SMM's design.

All functions of the smart memory module are thoroughly tested in the simulator to confirm correctness. All eight instructions are confirmed to work correctly under all conditions. The real-time nature of the garbage collector, read barrier, and object allocator are checked and work properly. The next section in this chapter deals with the timing results from each of these tests. An example of one of the many tests that is used to prove the SMM's functionality is shown below in Figure 5.1

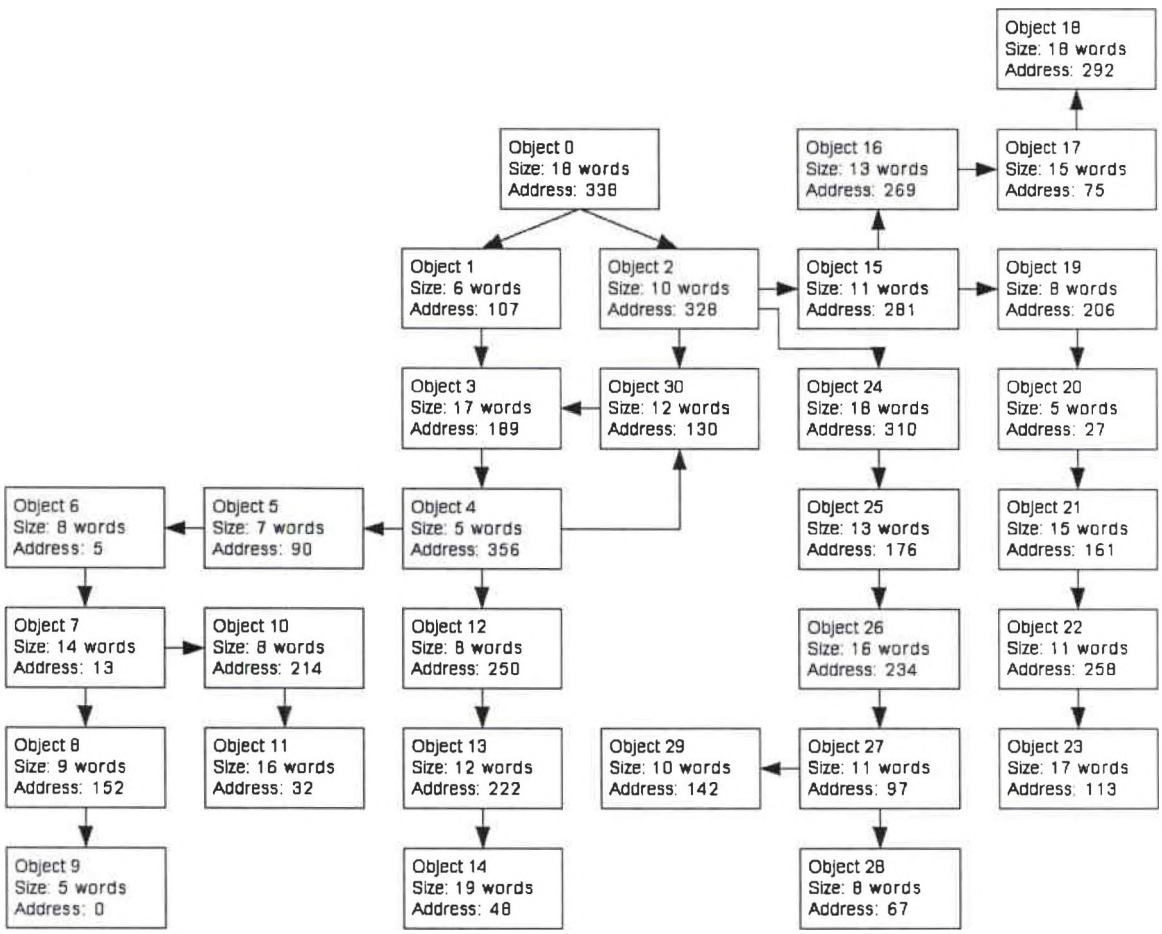

Figure 5.1: A Sample Heap

The figure shows a sample test heap that is used to test the garbage collection aspect of the smart memory module. All objects in the figure begin the collection cycle in the FromSpace and are gradually copied to the ToSpace. The sample heap contains one object that is referenced by the system stack and several descendant objects. The

objects on the heap contain varying sizes, memory locations, and number of references. Some objects are referenced only once while others have several references. There is one circular data structure that is uncollectible when using a reference counting algorithm. Figure 5.2 shows that progress of the Scan and Free pointers during this one test.

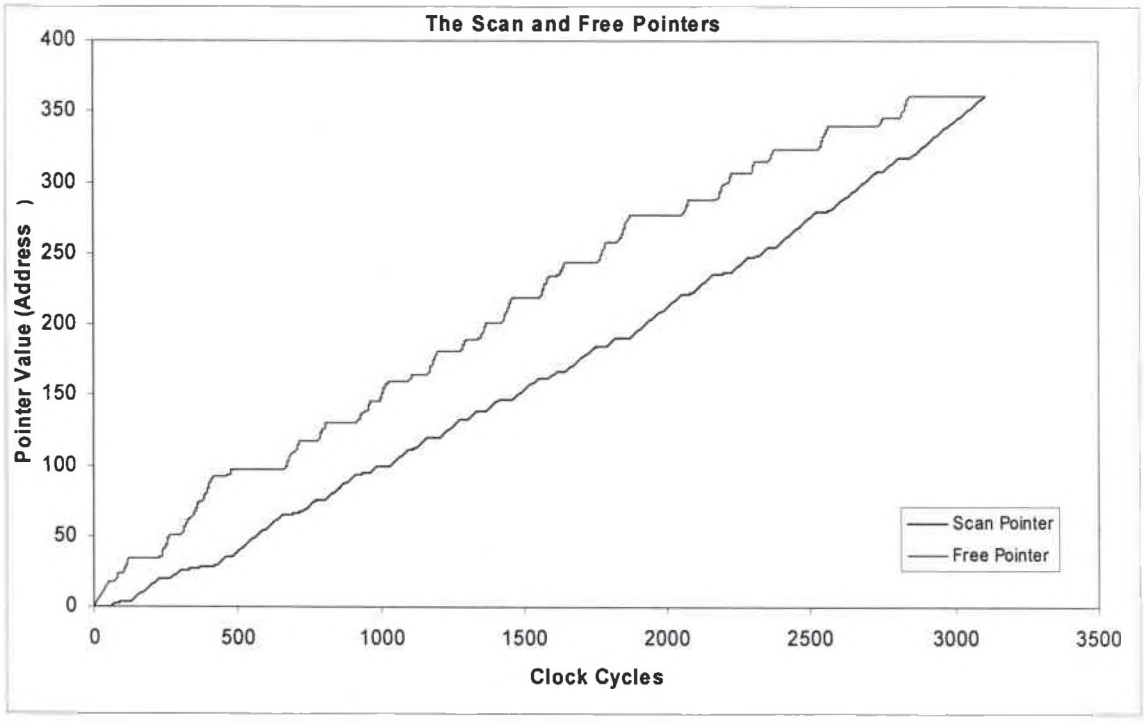

Figure 5.2: The Convergence of the Scan and Free Pointers

The Free pointer moves its reference ahead of the Scan pointer once the root object is copied to ToSpace. The Free pointer maintains this lead throughout the entire collection process. The Scan pointer only catches up with the Free pointer at the end of the memory cleaning routine. It is the equality of the Scan and Free pointers that informs the SMM of the collection cycle's completion. This experiment's results are generated by the Modelsim environment. This level of detail, such as the Scan and Free pointer values, is not possible without the use of the simulator.

5.2 Timing Results

The section above discusses how the garbage collector is tested to verify correctness. Once this process is done it is possible to see how fast the smart memory module actually runs. The timing of every instruction that is described in Chapters 3 and 4 is analyzed in detail below.

5.2.1 Garbage Collector

The garbage collection process is obviously the most important function to analyze for timing results. Without it there is no purpose for having the smart memory module. As mentioned in previous chapters, the garbage collection procedure is split into two parts. These two modules are the stack collector and the heap collector. The stack collector is analyzed first for timing results.

5.2.1.1 Stack Collector Results

The module traces the system stack, from bottom to top, for references to the FromSpace. There are three possible data types that the stack collector may encounter: plain data, references, and object headers. The stack collector takes no action if it encounters either plain data or object header data. The module just proceeds to the next data entry. The collector takes six clock cycles to retrieve, analyze, and then proceed to the next memory entry when it encounters these two data types. The execution time for the third data type, references, depends on what and where the reference points too. The module takes six clock cycles if the reference is to ToSpace. The stack collector only needs eleven clock cycles if the reference is to a forwarding pointer. During this time the module updates the original reference to the new ToSpace location. The time that is required if the reference is to an object in FromSpace varies on the size of the object. Equation 5.1 describes this variable time requirement.

$$ClockCycles = \begin{cases} FLOOR(ObjectSize-1,4)\times 8 + N + MOD(ObjectSize-1,4) + 4, & (ObjectSize-1)\,is\ not\ div.by\ 4 \\ FLOOR(ObjectSize-1,4)\times 8 + N, & (ObjectSize-1)\,is\ div.by\ 4 \end{cases} \quad (5.1)$$

[3]

The variable ObjectSize in the equation above is the size of the referenced FromSpace object in words. The N variable is a constant that represents the overhead cost that is required to copy each object regardless of size. The value of N is thirteen in this implementation of the SMM. Figure 5.3 plots the equation along a wide range of object sizes.

[3] The function FLOOR is equivalent to rounding the number down to the nearest whole value. The function MOD returns the modulus of the function.

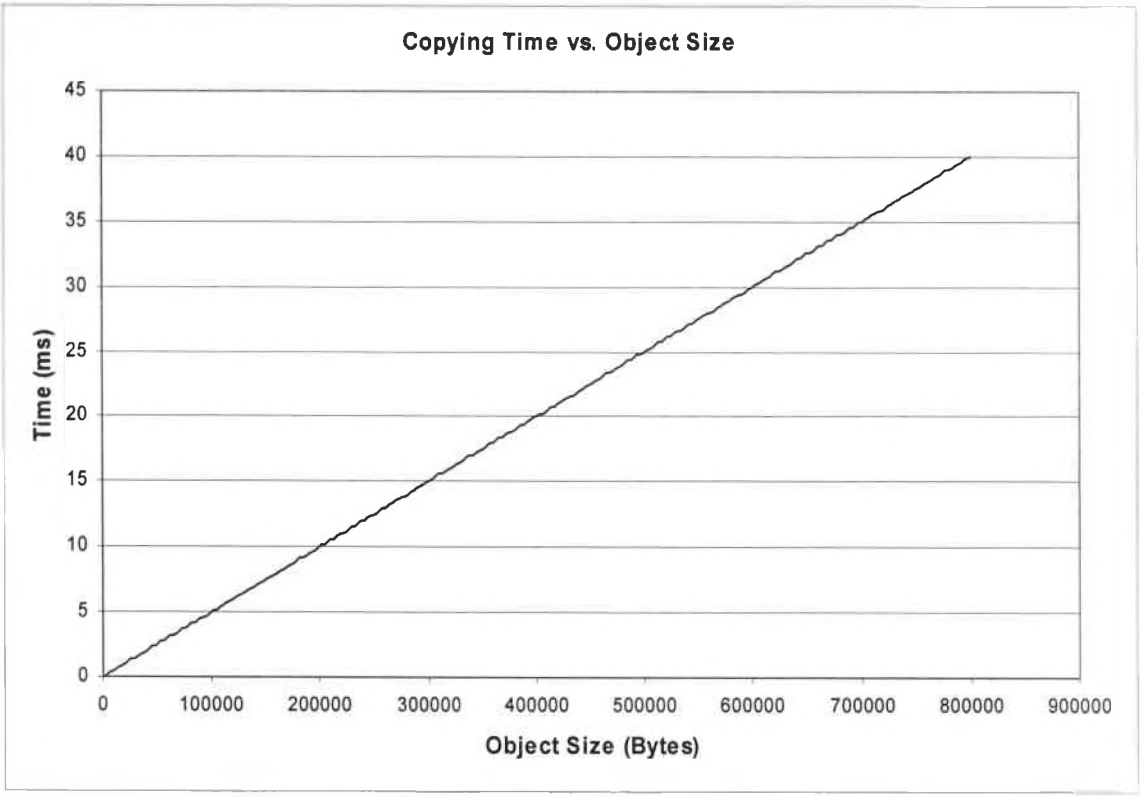

Figure 5.3: The Time Cost for Collecting Objects in the Stack Collector

The times for the figure are based on a 10MHz system clock. The figure's results show that the SMM is easily capable of collecting objects within a short period of time. The cost of copying rises as expected with an increase in the object's size. Figure 5.4 shows that the SMM collects a 100KB object in about 5ms. Table 5.1 shows a summary of the timing results for the stack collector

Table 5.1: The Timing Results for the Stack Collector

Data Type	# of Clock Cycles	Time (μs) @10MHz
Plain Data	6	0.6
Object Header	6	0.6
Reference – ToSpace	6	0.6
Reference – Forwarding Pointer	11	1.1
Reference – FromSpace	See Equation 5.1	See Figure 5.4

5.2.1.2 The Heap Collector's Results

The heap collector's results are very similar to the stack collector's. The purpose of the heap collector is to scan objects in ToSpace for descendants that still exist in FromSpace. The heap collector uses the Scan pointer to analyze memory words and the

Free pointer to copy objects to ToSpace. There are three possible data types that the collector may encounter while scanning the heap. These are plain data, object headers, and references. The system takes no action if it encounters either plain data or object headers during collection. The SMM requires six clock cycles to analyze and then pass over this data. The heap collector's reaction to references depends on what is being referenced. If the reference is to ToSpace then the module treats it as plain data and moves on to the next entry. This process takes six clock cycles. The system requires eleven clock cycles if the reference is to a forwarding pointer in FromSpace. The module corrects the original reference's value during this period. Finally, the amount of time that is needed if the reference is to an object in FromSpace varies. Equation 5.2 describes the time that is required as a function of object size.

$$Clock\ Cycles = \begin{cases} FLOOR(ObjectSize-1,4)\times 8 + N + MOD(ObjectSize-1,4) + 4, & (ObjectSize-1)\ is\ not\ div.by\ 4 \\ FLOOR(ObjectSize-1,4)\times 8 + N, & (ObjectSize-1)\ is\ div.by\ 4 \end{cases} \quad (5.2)$$

The equation is the same as the stack collector's copying time equation. The variable N is equal to thirteen for the heap collector. Figure 5.4 plots Equation 5.2 as the size of the object increases.

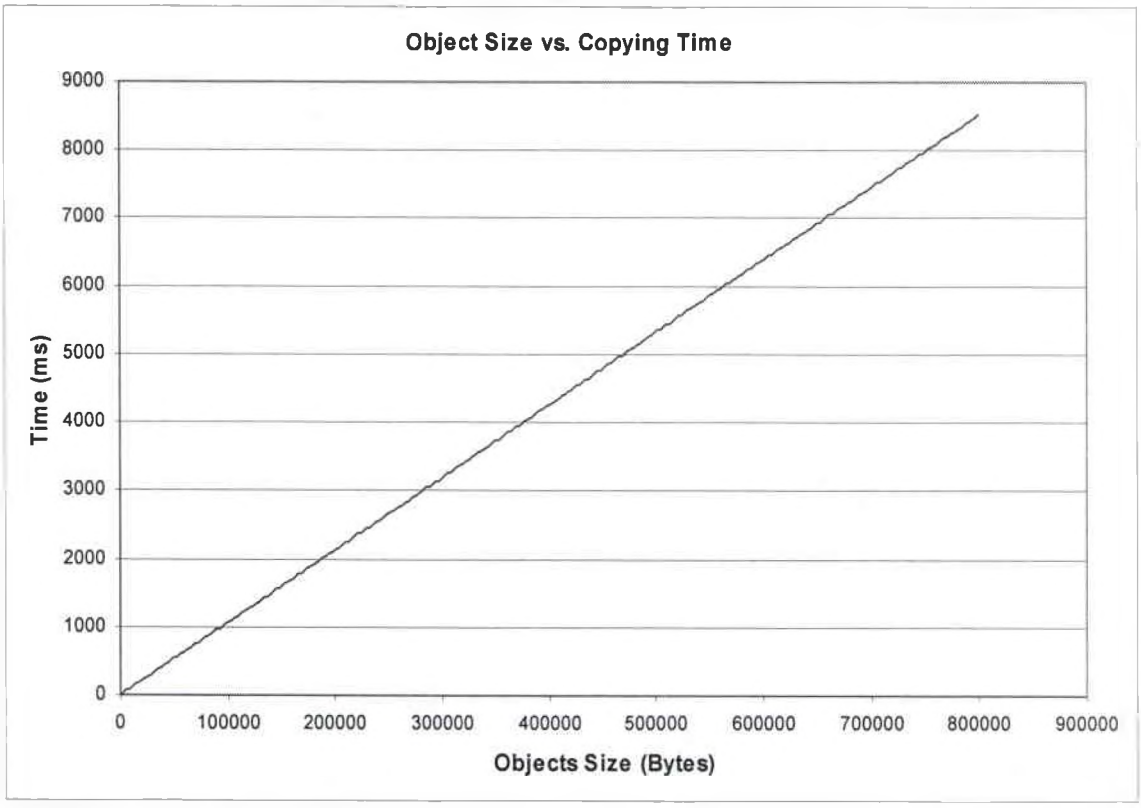

Figure 5.4: The Time cost for Collecting Objects in the Heap Collector

The results that are shown in Figure 5.4 are the same as Figure 5.3. They show that the SMM collects small objects at an extremely quick speed, and even collects larger objects within a reasonable amount of time. The figure shows that an object with four hundred bytes of data takes only 21.2μs to collect. The collector only requires milliseconds when the object size becomes very large, and the SMM is not even designed for these large objects. A summary of the heap collector's timing results is shown below in Table 5.2

Table 5.2: The Timing Results for the Heap Collector

Data Type	# of Clock Cycles	Time (μs) @10MHz
Plain Data	6	0.6
Object Header	6	0.6
Reference – ToSpace	6	0.6
Reference – Forwarding Pointer	11	1.1
Reference – FromSpace	See Equation 5.2	See Figure 5.4

5.2.2 Read Barrier

The read barrier enforces two rules that prevent the mutator from catastrophically damaging the heap during its execution. These rules are never letting a black object[4] reference a white object and preventing the heap from accidentally having two locations for one object. The SMM's read barrier activates whenever the mutator attempts to read data from the heap. There are two HDL modules in the SMM that enforce the barrier. One protects the system stack data and the other protects the heap data. The stack read barrier is the first to be examined for results.

5.2.2.1 Stack Read Barrier's Results

This barrier is activated whenever the mutator tries to 'pop' data of the system stack. The requested data is examined to see if there are improper references being read. The stack read barrier comes across three data types. Two of these types, plain data and object headers, require no action. The SMM uses only nine clock cycles to check these data types for barrier compliance and then return the information to the mutator. The stack is also 'popped' during this process. The third type of data, references, requires a more complex reaction. The memory module uses nine clock cycles if the reference is to an object in the ToSpace. Thirteen clock cycles are necessary if the stack read barrier encounters a reference to a forwarding pointer. The barrier updates the reference to the new ToSpace location and 'pops' the system's stack during this period. The stack's read barrier requires a variable amount of time if the reference is to a FromSpace object. This variable amount of time is dependent on the object's size. Equation 5.3 describes this dependence on object size.

$$ClockCycles = \begin{cases} FLOOR(ObjectSize-1,4)\times 8 + N + MOD(ObjectSize-1,4) + 4, & (ObjectSize-1)\ is\ not\ div.by\ 4 \\ FLOOR(ObjectSize-1,4)\times 8 + N & ,(ObjectSize-1)\ is\ div.by\ 4 \end{cases}$$

(5.3)

The main difference between this equation and the previous two is the value of the variable N. N is equal to fourteen in this case. Figure 5.5 shows equation 5.3 as a variable of object size with the system clock running at 10MHz.

---

[4] The color black refers to objects that are found to be live and do not need to be revisited by the collector. The color gray means an object is live but needs to be scanned by the collector for descendants. White objects have yet to be reached by the collector, and all white objects at the end of a collection cycle are declared garbage.

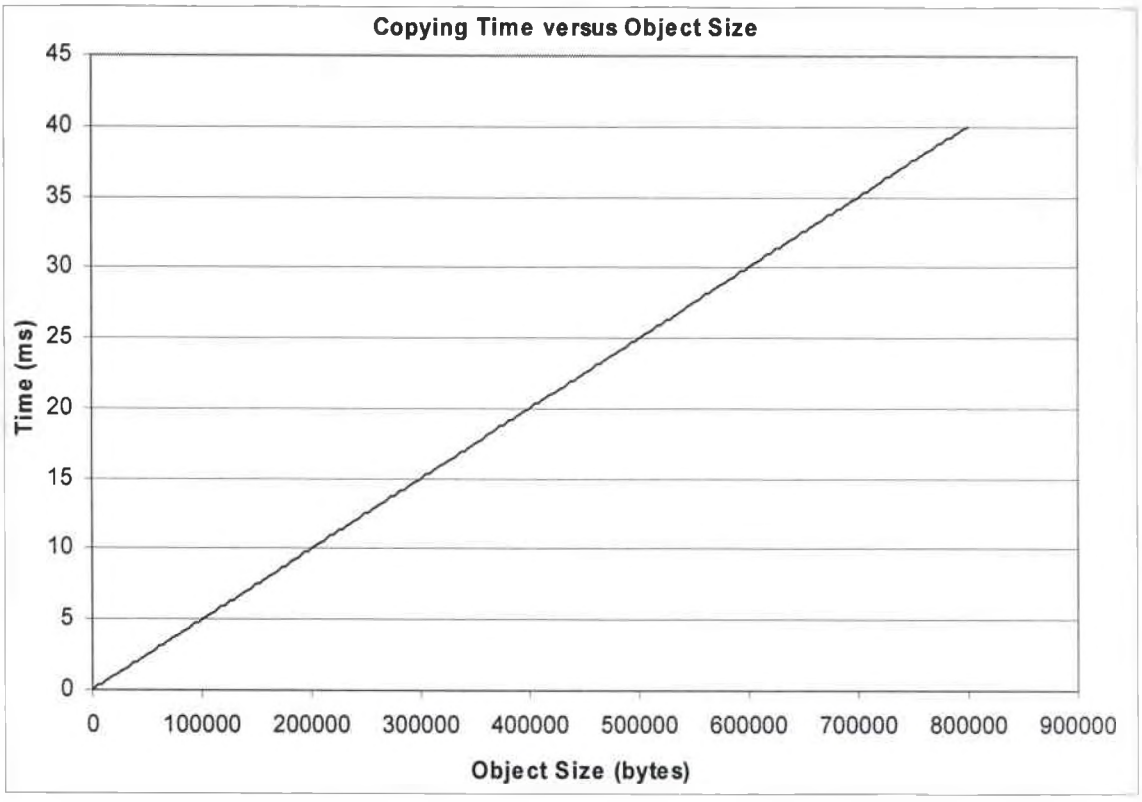

Figure 5.5: The Time Cost for Collecting Objects in the Stack Read Barrier

The figure shows that the stack read barrier fetches memory data at a fast rate even for large objects. An object with a size of one hundred thousand bytes only requires 5ms to read if the object is still in FromSpace. Table 5.3 shows a summary of the stack read barrier's results.

Table 5.3: The Results of the Stack Read Barrier

Data Type	# of Clock Cycles	Time (μs) @10MHz
Plain Data	9	0.9
Object Header	9	0.9
Reference – ToSpace	9	0.9
Reference – Forwarding Pointer	13	1.3
Reference – FromSpace	See Equation 5.3	See Figure 5.5

5.2.2.2 The Heap Read Barrier's Results

The results of the heap read barrier are now discussed. The three types of data that the heap read barrier encounters are plain data, object headers, and references. The SMM allows the mutator to read plain data and object headers without incident. These types of read commands require nine clock cycles to execute. The number of clock

cycles that are needed to read a reference depends on what is being referenced. The SMM requires only nine clock cycles to read a reference to an object in ToSpace. A total of fourteen clock cycles are necessary if the reference is to a forwarding pointer in FromSpace. The collector updates the original reference to the new ToSpace location during this time. The time that is required by smart memory module to read a reference to an object in FromSpace varies based upon the object's size. Equation 5.4 describes the time requirement as a variable of object size.

$$Clock\,Cycles = \begin{cases} FLOOR(ObjectSize-1,4)\times 8 + N + MOD(ObjectSize-1,4) + 4, & (ObjectSize-1)\,is\,not\,div.by\,4 \\ FLOOR(ObjectSize-1,4)\times 8 + N, & (ObjectSize-1)\,is\,div.by\,4 \end{cases}$$

(5.4)

The equation above is the same as the first three equations in this chapter. The main difference is that the variable N, which represents copying overhead, is fifteen is this case. Figure 5.6 shows this equation as the object size ranges from four bytes to eight-hundred thousand bytes.

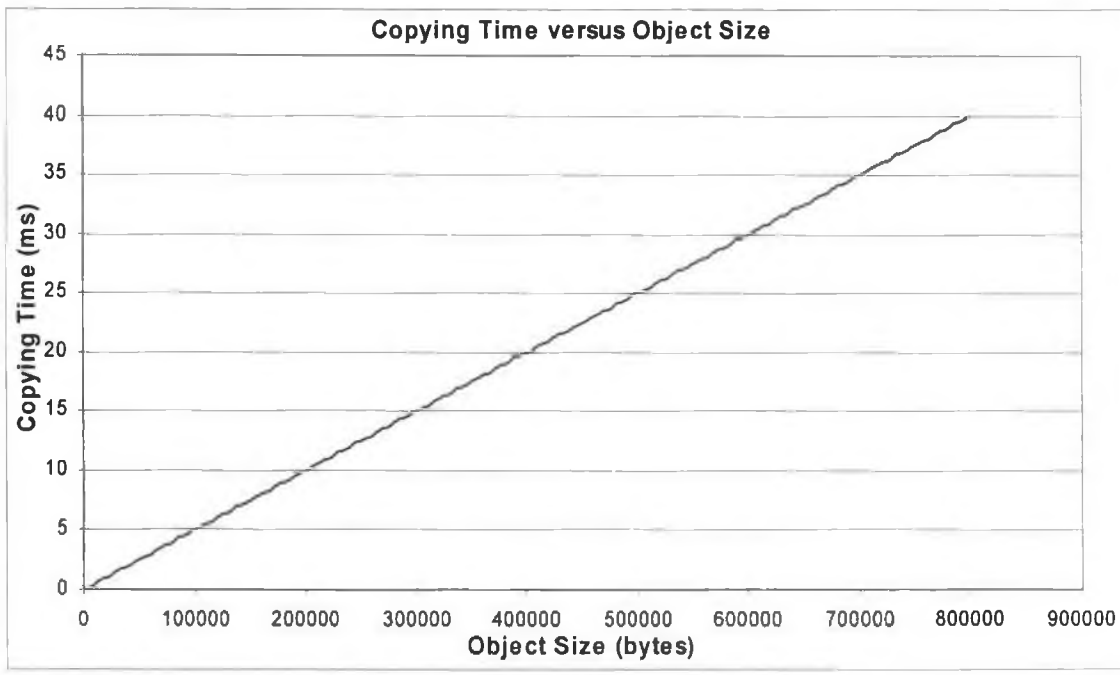

Figure 5.6: The Time Cost for Collecting Objects in the Heap Read Barrier

The results for the heap read barrier are similar to the other collector and read barrier results. An object of four hundred bytes needs only 21.4μs to collect and return a new reference to the mutator. Table 5.6 shows a summary of the heap read barrier's results.

Table 5.4: The Results for the Heap Read Barrier

Data Type	# of Clock Cycles	Time (μs) @10MHz
Plain Data	9	0.9
Object Header	9	0.9
Reference – ToSpace	9	0.9
Reference – Forwarding Pointer	14	1.4
Reference – FromSpace	See Equation 5.4	See Figure 5.6

5.2.3 Miscellaneous Results

This section focuses on the timing results for the remainder of the smart memory module's functions. These functions are object allocation, memory writes, stack writes, setting the GC timer, and setting the free space limit.

5.2.3.1 Object Allocation Results

Fast object allocation is one of the major benefits of using Baker's copying collector. The procedure is as cheap as pushing data onto a stack. This savings is because the collector automatically compacts data during the collection cycle leaving a continuous area of free memory. The New pointer references this free space. The memory module only needs to look at the New pointer to determine the new object's location. The worst case time for object allocation is three clock cycles, and the most common time is two clock cycles. The worst case timing occurs when the ToSpace does not have enough free memory for the new object and the collector must flip the ToSpace and FromSpace. The two clock cycle allocation happens when the current ToSpace does have enough free memory for the new object.

5.2.3.2 Memory Writes

The garbage collector takes no special action when the mutator attempts to write to memory. This response is because the smart memory module enforces a read barrier instead of a write barrier. Memory writes take two clock cycles.

5.2.3.3 Stack PUSH

Pushing data onto the system stack in the SMM requires very little time. The current SMM needs two clock cycles to put data on top of the stack.

5.2.3.4 Setting the GC Timer and Free Space Limit

The smart memory module only requires one clock cycle to set either the GC timer or the Free Space register.

5.2.4 Results Summary

The timing results for the five instructions above show that the SMM provides a fast and effective means to manage the heap memory. The timing for the object allocation command is especially good. Only three clock cycles are needed to allocate a new object memory space on the heap. Table 5.5 lists a summary of all the results.

Table 5.5: The Smart Memory Module's Miscellaneous Timing Results

Data Type	# of Clock Cycles	Time (μs) @10MHz
Object Allocation – Best Case	2	0.2
Object Allocation – Worst Case	3	0.3
Memory Write	2	0.2
Stack PUSH	2	0.2
Setting GC Timer	1	0.1
Setting Free Space Limit	1	0.1

5.3 The Smart Memory Module versus the Garbage Collecting Memory Module

This paper uses Kelvin Nilsen's garbage collecting memory module (GCMM) as a key reference in the design of the smart memory module. Nilsen embeds his design in a memory module and uses independent memory buses to decrease copying time. Both of these ideas are used in the SMM. The main difference between the two designs is that Nilsen uses a RISC processor to control his collection process while the SMM uses custom hardware for the same task. It therefore makes sense to compare the timing results of the two memory modules. Figure 5.3 shows the results of this comparison.

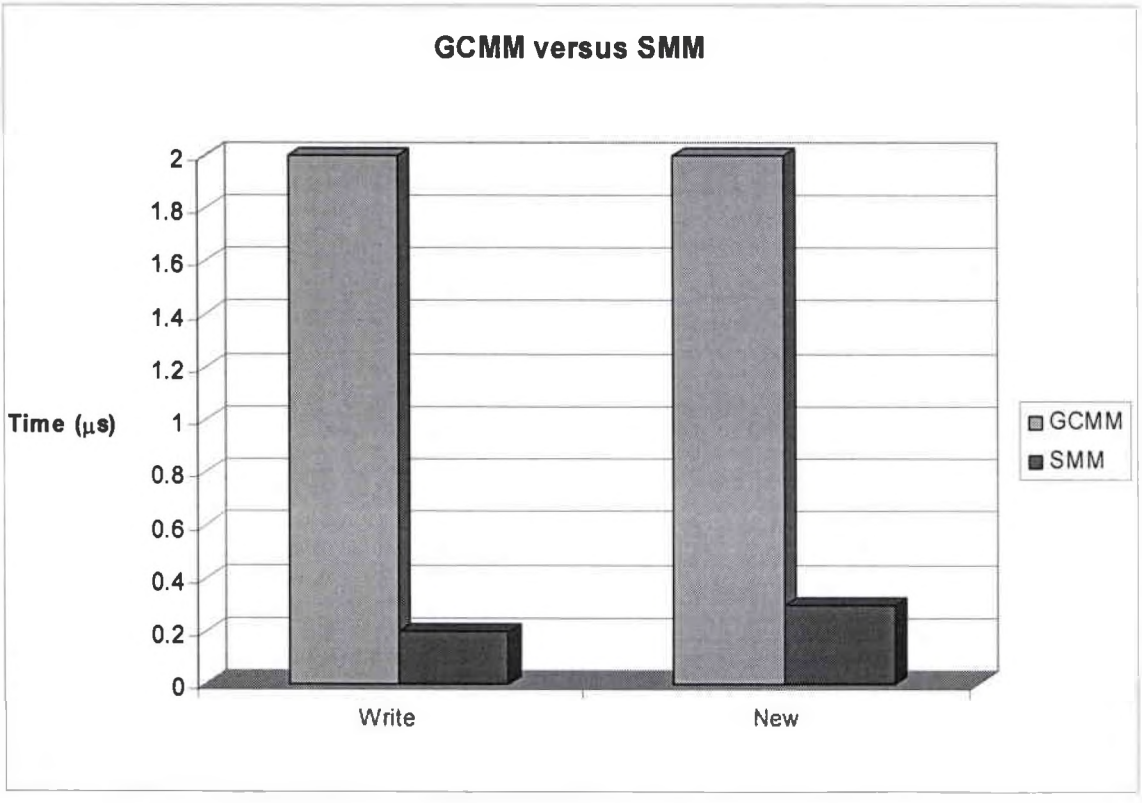

Figure 5.7: The GCMM versus the SMM

The figure shows that the embedded hardware performs at much higher speeds when even operating at a 10MHz clock. The first prototype of the SMM dramatically outperforms the GCMM in both write and new commands. The write instruction is ten times faster and the allocation instruction is more than seven times faster. The read command is not graphed in the figure because Nilsen does not give adequate data to make a valid comparison.

5.4 Summary

The results in this chapter show that the smart memory module performs real-time garbage collection correctly and expeditiously. Most of the SMM's operations require only a few clock cycles to complete. In particular, the memory allocation command needs at worst only three clock cycles. This outcome is a tremendous improvement over other memory allocation commands that search the entire memory for a suitable block of data for the new object. The time that is required for garbage collection is fast, bounded, and predictable. The collection time of the SMM increases linearly with the size of the

object. A four-hundred byte object uses only 21.2μs running at 10MHz to collect. Finally, the smart memory module performs at a much faster rate than Nilsen's GCMM. The current version of the SMM is primarily a proof of concept design. The speed of the hardware, the efficiency of the design, and the quality of the collection algorithms will increase in later generations of the hardware.

Chapter 6

Future Improvements to the Smart Memory Module

The smart memory module that is designed in this paper offers a fast, bounded, and predictable means to collect heap memory. The results of the module show that it is much faster than even Nilsen's memory module that uses an internal RISC processor. However, there are areas of improvement for the SMM. This chapter explores two avenues of future research that may be implemented in later versions of the SMM. One topic explores ways to replace the read barrier with a write barrier. The other topic explores methods that increase the copying speed of the collector.

6.1 A Handle Pool for the Collector

The copy garbage collector that is implemented in this thesis uses a read barrier to prevent the mutator from modifying the heap catastrophically during the collection cycle. Chapter 2 discusses this problem in detail. The other real-time collectors use a write barrier. They are able to use such a barrier because their collection algorithms modify heap data but do not move the actual objects.

Both write and read barriers cost the system time to execute. It is known that the most efficient barrier for a particular system is the one that is used the least. A read barrier is most efficient for a system that performs few read commands. A write barrier is more efficient in a system with more read commands than write commands. Research has been done that shows that most systems issue far more read commands than write commands [1]. That is why the non-copying collectors use a write barrier over a read barrier. Both barriers prevent the mutator from mangling the heap, but the write barrier is usually chosen over the read barrier because it is invoked less often.

Unfortunately, the copy collectors do not have a choice in the type of barrier to use. That is because the copy collector moves objects during the collection cycle instead of just modifying the data. The copying leads to the problem of accidentally having multiple locations for a single object. This phenomenon is described in Chapter 2. Write barriers only prevent the mutator from maliciously modifying data. Read barriers prevent against dangerous data changes and from having multiple locations for one object. That is why the copy collectors must use a read barrier instead of a write barrier.

It is desirable to use a write barrier instead of a read barrier for a copying garbage collector. That is because read commands should be much more numerous than write commands. A write barrier reduces the impact that the collector has on the system during mutator execution. However, the previous paragraphs show that the current copy collector cannot use a write barrier. The collector needs to be modified so that the objects appears static to mutator in order to use a write barrier. One way to accomplish this task is to use a handle pool.

The handle pool is a block of memory that acts as a buffer between objects on the heap and references to them. Each object on the heap is assigned a word in the handle pool. This allocated word's address in memory is static and never changes. No reference directly references an object's location on the heap. All references point to the object's assigned memory word on the handle pool. The handle pool contains the object's current location on the heap. The benefit of a handle pool is that all references are to a static location and only the handle pool's value needs updating when an object is relocated. The static references to the handle pool eliminate the problem of accidentally having multiple locations for one object, and allows for a write barrier instead of a read barrier. The figure below shows an example of the handle pool.

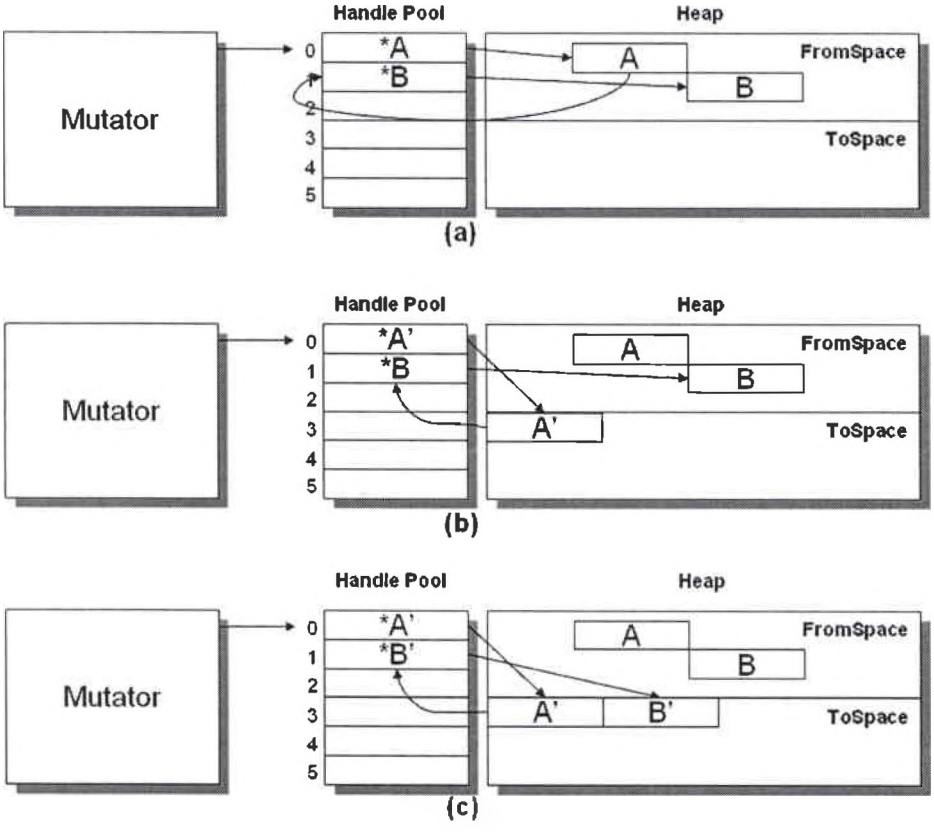

Figure 6.1: An Example of the Handle Pool

Figure 6.1(a) shows that there are two objects on the heap and both objects are referenced in the handle pool. The mutator's root references Object A. Object A then references Object B. Figure 6.1(a) also shows that all references first point to the handle pool and the handle pool then points to the object's actual location. Object A is copied to the ToSpace in Figure 1(b). The mutator's root reference to Object A remains unchanged. It has no idea that Object A's location in the heap has changed. Only the reference to Object A in the handle pool is updated to represent the new location. Figure 6.1(c) shows Object B being collected and moved to the ToSpace and the handle pool reflects this change. It is important to note that the references to the handle pool remain static throughout the entire example. Only the values of the handle pool change when an object is moved. The static location of handle pool values eliminates the possibility of multiple locations for one object. This benefit means that a write barrier can be used for the copy collector instead of a read barrier.

The use of a write barrier improves the performance of the system over that of a read barrier, but the use of a handle pool does produce some overhead costs. The handle pool slows down access to objects on the heap. The system must first retrieve the value of the handle pool and then access the object. Systems without handle pools have direct access to objects. However, the delay that is caused by indirect referencing is still minimal compared to the cost of executing a read barrier.

Handle pools also slow down object allocation time. This problem is the handle pool's largest drawback. Each object is assigned a word in the handle pool upon creation. The system must search for a free handle pool word each time a new object is allocated. The worst case scenario is that the entire handle is scanned for a free space before one is found. The handle pool might be thousands of words long and the delay that is caused by allocation is unacceptable to a real-time system.

One way around the handle pool allocation problem is to implement the handle pool in a way similar to Baker's Treadmill collector that is described in Chapter 2 [12]. The handle pool constructs a doubly linked circular list upon initialization. Each handle pool cell contains three memory words. Two words are used to link the list and the remaining word is used as the reference to the heap. Figure 6.2 shows this structure for the handles.

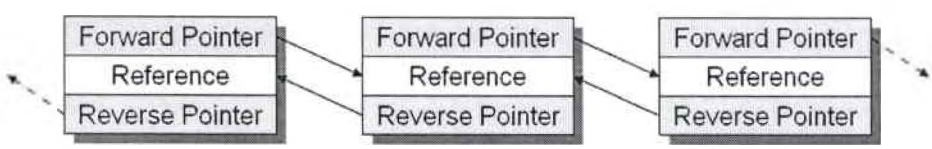

Figure 6.2: The Handle Pool Cells

Figure 6.2 demonstrates that each handle needs three words of memory for one reference to the heap. The handle pool is then organized by two pointers. These pointers are Allocate and Unoccupied. Figure 6.3 shows this organization.

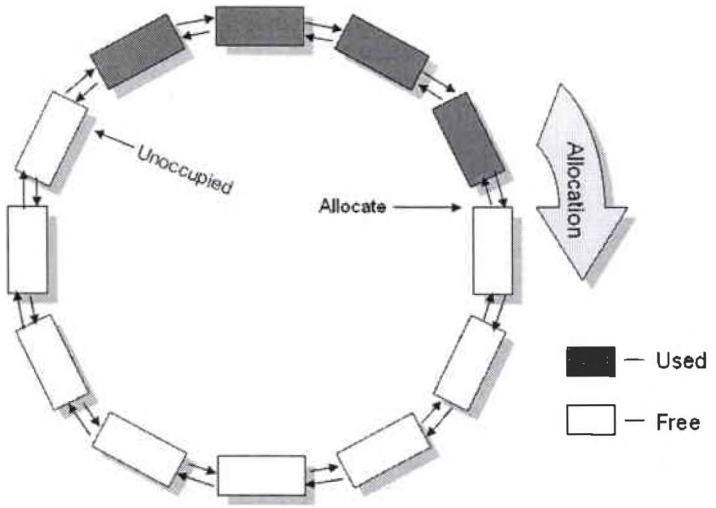

Figure 6.3: The Handle Pool Treadmill

The two pointers in the figure above separate the free and used locations in the handle pool. All cells between the Unoccupied and Allocate pointers, moving clockwise from Unoccupied, are being used. The remaining cells are free for allocation. New handles are assigned at the Allocate pointer. The pointer is then advanced clockwise to the next free cell in the list. This allocation is much faster and predictable than the other method searching handle pool for a free word. Every allocation from the handle pool takes an equal amount of time and costs about as much as stack allocation. This speed is because the system allocates the new handles at the Allocate pointer and does not need to search for a free cell.

Deallocation of handle pool cells is also simple and quick. The pointers for the deleted reference cell are changed so that the cell is in the free portion of the list. The forward and reverse pointers of the cell's surrounding the handle's old location in Used space are modified to account for its absence. A total of six pointers are written to during this process. This is the only time that the linked list pointers are modified. The handle pool is out of space when the Unoccupied and Allocate pointers are equal. It is important to note that no data is actually moved during allocation and deallocation of handles. Only the values of the Unoccupied and Allocate pointers along with each handle pool cell's forward and reverse pointers are modified. Each handle pool cell location is static in memory and provides a fixed reference. The fixed reference allows for the write barrier

instead of the read barrier. The only drawback to this algorithm is that it requires three times the amount of memory for each reference.

6.2 Increasing Copy Efficiency

The largest problem confronting the use of Baker's copy collector in real-time programs is the cost of moving objects from the FromSpace to the ToSpace. The cost for copying objects is minimal when they are small but increases linearly as they grow in size. The SMM's results in Chapter 5 attest to this fact. Increasing the speed of the memory helps alleviate the problem, but this solution necessitates waiting on memory manufactures to improve their product.

An alternative solution to the problem is to increase the memory word size using current memory technology. The current SMM design consists of two separate memory modules with independent buses. This configuration is shown in the figure below.

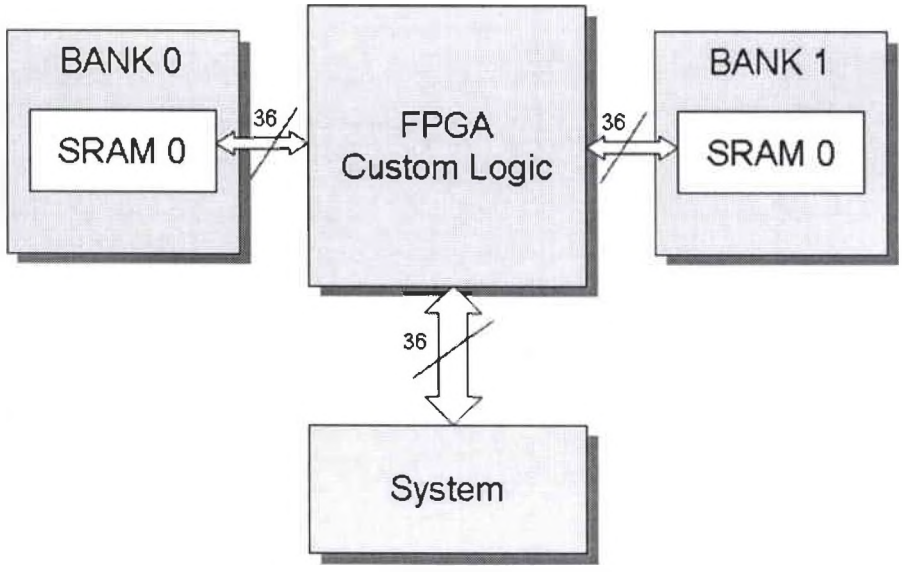

Figure 6.4: The Current SMM Architecture

The two memory modules contain the ToSpace and FromSpace. The independent memory buses allows for a faster copying time than if there was only one shared bus. This benefit is because the SMM can read from the FromSpace and Write to the ToSpace in parallel. The current width of the memory buses is thirty-six bits. This bus width limits the SMM to copying one object word at a time. The copying time is cut

dramatically if the SMM memory word is increased by multiples of thirty-six. The diagram below shows a potential layout of this new system.

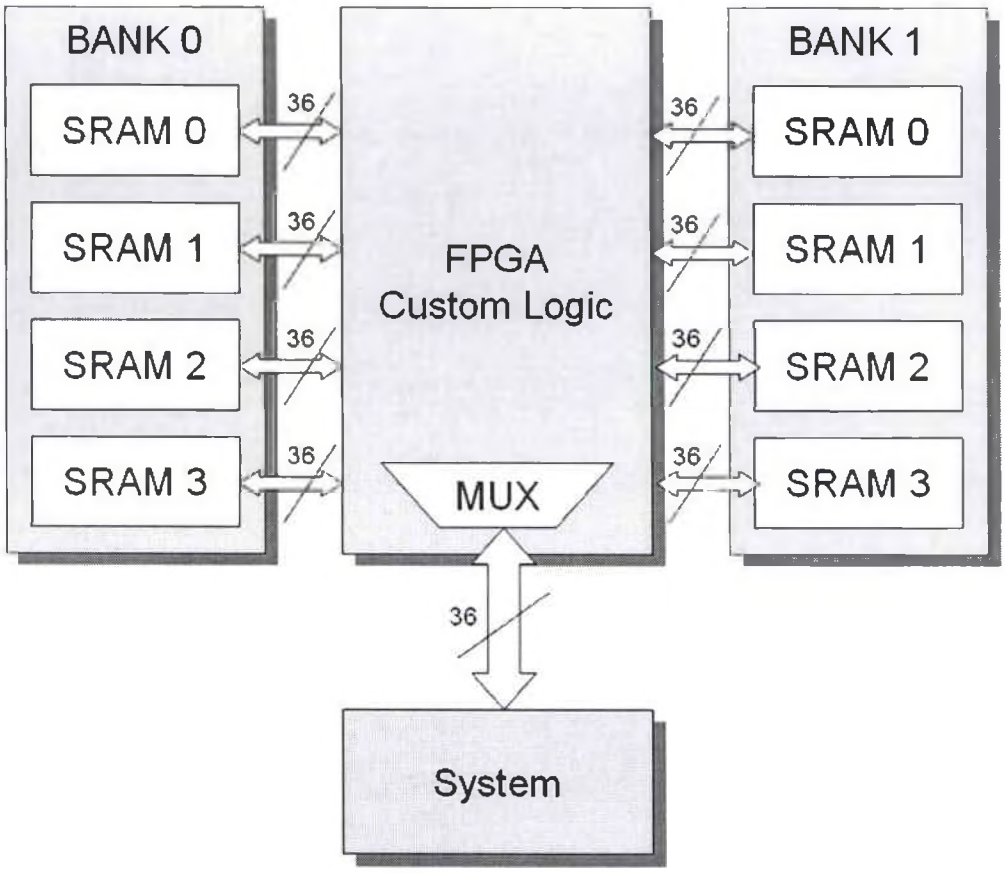

Figure 6.5: A SMM with a Wide Memory Word

The example above shows just one possible configuration of the wide memory word structure for the SMM. The new memory word width is one hundred forty four bits. This configuration allows for the possibility of copying four times the amount of data over the original smart memory module. Up to a hundred a forty-four bits may be copied in one clock cycle instead of thirty-six bits. The new SMM with the wide memory word is almost four times faster than the original design, and the new system just uses six more memory module to achieve this improvement.

The mutator is still able to access the memory through a traditional thirty-six bit wide word that is individually addressable. If the mutator wishes to read a piece of data then it sends the SMM the proper address for the thirty-six bit wide section of data. The

SMM uses the sixteen most significant address bits[5] to retrieve the correct 144-bit word from memory. The remaining two least significant bits are used by the multiplexer that is shown above to fetch the correct thirty-six bits from the larger 144-bit word. The mutator is completely unaware that the SMM uses an internal 144-bit bus. The system program accesses the memory using the same address and data width. Mutator writes to memory are handled in a similar way. The sixteen most significant bits are used to address the extra-wide 144-bit word. The remaining two least significant bits used by a multiplexer to write enable only the correct thirty-six bit memory module.

The main improvement of the wide memory word SMM is the speed increase. The copying collection algorithm runs at roughly four times the speed as the original memory module. Even faster systems are possible if the internal memory word size is made larger. A faster garbage collecting algorithm means that the system can accommodate even larger objects. The second advantage to the new design is that it has no impact on how the mutator views the memory module. The mutator interacts with the new module in the same fashion as the old SMM design. The system program maintains a 36-bit wide data word. The 144-bit word only affects the internal workings of the SMM. The newer module is backwards compatible with systems that use an older SMM model. The final benefit is that the new design is possible with current memory technology. The SMM with a wide memory word does not have to wait on faster memory technology to increase its speed. The new design is implemented by using six more memory modules that are identical to the two in the older SMM design. It is possible to currently build this newer and faster design.

[5] It is assumed that the memory addresses are still eighteen bits long like in the original smart memory module design.

References

1. Jones, Richard. Lins, Rafael. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons. Chichester. 1996.
2. McCarthy, John. "Recursive Functions of Symbolic Expressions and their Computation by Machines." Communications of the ACM, 3:184-195, 1960.
3. Boehm, Hans-Juergen. Weiser, Mark. "Garbage Collection in an Uncooperative Environment." Software Practice and Experience, 18(9):807-820, 1988.
4. Lins, Rafael D. "Cyclic Reference Counting with Lazy Mark-Scan." Information Processing Letters, 44(4):215-220, 1992.
5. Weizenbaum, J. "Symmetric List Processor." Communications of the ACM, 6(9):524-544, September 1963.
6. Cheney, C. J. "A Non-Recursive List Compacting Algorithm." Communications of the ACM, 13(11):677-678, November 1970.
7. Nilsen, Kelvin. "Reliable real-Time Garbage Collection of C++." OOPSLA Workshop on Garbage Collection in Object-Oriented Systems. 1993.
8. Withington, P. T. "How Real is "Real-Time" GC". OOPSLA Garbage Collection Workshop. 1991.
9. Yuasa, Taichi. "Real-Time Garbage Collection on General Purpose Machines." Journal of Software and Systems, 11(3):181-198, 1990.
10. Dijkstra, Edsgar W., Lamport, L., Martin, A. J., Scholten, C.S., and Steffens, E. F. M. "On-the-fly Garbage Collection: An Exercise in Cooperation." Lecture Notes in Computer Science, No. 46. Springer-Verlag, New York, 1976.
11. Baker, Henry G. "List Processing in Real-Time on a Serial Computer." Communications of the ACM. 21(4):280-94. 1978.
12. Baker, Henry G. "The Treadmill, Real-Time Garbage Collection without Motion Sickness." ACM SIGPLAN Notices, 27(3). March 1992.

13. Nilsen, Kelvin D., "Cost-Effective Hardware-Assisted Real-Time Garbage Collection." Workshop on Language, Compiler and Tool Support for Real-Time Systems, 1994.
14. Smith, Douglas J. HDL Chip Design: A Practical Guide for Designing, Synthesizing, and Simulating ASICs and FPGAs using VHDL or Verilog. Doone Publications. Madison, Al. 1996.
15. Virtex-II Pro Platform FPGA Handbook. Xilinx. San Jose California. 2002.
16. SLAAC-1V User VHDL Guide, Release 0.3.1, University of Southern California Information Sciences Institute, 2000.
17. SLAAC-1V SDK User's Manual, Release 0.3.1, University of Southern California Information Sciences Institute, 2000.